Cray Assembly Language (CAL) for Cray X1™ Systems Reference Manual

S–2314–51

New Features

Cray Assembly Language (CAL) for Cray X1™ Systems Reference Manual S–2314–51

Clarified description of Cray X1 Instruction Set and Encoding notations (Section 7.1, page 103), reversed the order of the CAL Pseudo Instruction Descriptions and Cray X1 Instruction Set and Encoding chapters, and corrected technical errors in Cray X1 Instruction Set specifications.

Record of Revision

Version	*Description*
1.0	August 19, 2002 Draft version to support the Programming Environment 4.1 release.
1.1	June 2003 Supports the Cray Assember 1.1 and Programming Environment 5.0 releases.
1.2	October 2003 Supports the Cray Assember 1.2 and Programming Environment 5.1 releases.

Contents

Page

Page

Figures

Page

Tables

Preface

This manual supports the Cray Assembler 1.1 release running on Cray X1 systems. This preface describes how to access Cray documentation and error message explanations, interpret our typographical conventions, order Cray documentation, and contact us about this document.

Accessing Cray Documentation

Each software release package includes the CrayDoc documentation system, a collection of open-source software components that gives you fast, easy access to and the ability to search all Cray manuals, man pages, and glossary in HTML and/or PDF format from a web browser at the following locations:

- Locally, using the network path defined by your system administrator
- On the Cray public web site at:

  `http://www.cray.com/craydoc/`

All software release packages include a software release overview that provides information for users, user services, and system administrators about that release. An installation guide is also provided with each software release package. Release overviews and installation guides are supplied in HTML and PDF formats as well as in printed form. Most software release packages contain additional reference and task-oriented documentation, like this document, in HTML and/or PDF formats.

Man pages provide system and programming reference information. Each man page is referred to by its name followed by a number in parentheses:

`manpagename (n)`

where *n* is the man page section identifier:

1	User commands
2	System calls
3	Library routines
4	Devices (special files) and Protocols
5	File formats
7	Miscellaneous information
8	Administrator commands

Access man pages in any of these ways:

- Enter the `man` command to view individual man pages in ASCII format; for example:

  ```
  man ftn
  ```

  To print individual man pages in ASCII format, enter, for example:

  ```
  man ftn | col -b | lpr
  ```

- Use a web browser with the CrayDoc system to view, search, and print individual man pages in HTML format.
- Use Adobe Acrobat Reader with the CrayDoc system to view, search, and print from *collections* of formatted man pages provided in PDF format.

If more than one topic appears on a page, the man page has one primary name (`grep`, for example) and one or more secondary names (`egrep`, for example). Access the ASCII or HTML man page using either name; for example:

- Enter the command `man grep` or `man egrep`
- Search in the CrayDoc system for `grep` or `egrep`

Error Message Explanations

Access explanations of error messages by entering the `explain` *msgid* command, where *msgid* is the message ID string in the error message. For more information, see the `explain`(1) man page.

Typographical Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements.
variable	Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace *filename* with the name `datafile` in

	your program. It also denotes a word or concept being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on.
...	Ellipses indicate that a preceding element can be repeated.

Ordering Documentation

To order software documentation, contact the Cray Software Distribution Center in any of the following ways:

E-mail:
`orderdsk@cray.com`

Web:
`http://www.cray.com/craydoc/`

Click on the `Cray Publication Order Form` link.

Telephone (inside U.S., Canada):
1-800-284-2729 (BUG CRAY), then 605-9100

Telephone (outside U.S., Canada):
Contact your Cray representative, or call +1-651-605-9100

Fax:
+1-651-605-9001

Mail:
Software Distribution Center
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120-1128
USA

Reader Comments

Contact us with any comments that will help us to improve the accuracy and usability of this document. Be sure to include the title and number of the document with your comments. We value your comments and will respond to them promptly. Contact us in any of the following ways:

E-mail:
`swpubs@cray.com`

Telephone (inside U.S., Canada):
1-800-950-2729 (Cray Customer Support Center)

Telephone (outside U.S., Canada):
Contact your Cray representative, or call +1-715-726-4993 (Cray Customer Support Center)

Mail:
Software Publications
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120-1128
USA

Introduction [1]

The Cray X1 Assembly Language (CAL) is a symbolic language for Cray X1 systems. The Cray X1 assembler generates object code from CAL source code for execution on Cray X1 systems.

1.1 Capabilities

The primary capability of CAL is to express with symbolic machine instructions the full Cray X1 instruction set. This manual does not describe the machine instructions in detail but does give an overview of the instructions and lists the instructions (Chapter 7, page 103). The Cray X1 instruction set is described in detail in the *System Programmer Reference for Cray X1 Systems.*

In addition, CAL provides these capabilities:

- The free-field source statement format of CAL lets you control the size and location of source statement fields.
- With some exceptions, you can enter source statements in uppercase, lowercase, or mixed-case letters.
- You can assign code or data segments to specific areas to control local and common sections.
- You can use preloaded data by defining data areas during assembly and loading them with the program.
- You can designate data in integer, floating-point, and character code notation.
- You can control the content of the assembler listing.
- You can define a character string in a program and substitute the string for each occurrence of its micro name in the program by using micro coding.
- You can define sequences of code in a program or in a library and substitute the sequence for each occurrence of its macro name in the program by using macro coding.

1.2 Related Publications

The following documents contain additional information that may be helpful:

- *System Programmer Reference for Cray X1 Systems*
- *Cray Fortran Compiler Commands and Directives Reference Manual*
- *Fortran Language Reference Manual, Volume 1*
- *Fortran Language Reference Manual, Volume 2*
- *Fortran Language Reference Manual, Volume 3*
- *Cray C and C++ Reference Manual*
- Loader man page `ld`(1)

Cray X1 Instruction Set Overview [2]

This chapter gives an overview of the Cray X1 system instruction set. The Cray X1 system is an implementation of the Cray NV-1 instruction set architecture. The full instruction set is listed in Chapter 7, page 103 and is described in detail in the *System Programmer Reference for Cray X1 Systems.*

2.1 Data Types

The Cray X1 system has four fundamental data types:

- `W`: 32-bit integer (Word)
- `L`: 64-bit integer (Longword)
- `S`: 32-bit IEEE-754 floating-point (Single)
- `D`: 64-bit IEEE-754 floating-point (Double)

The default for the data type of an instruction is `L`. The other data types are usually specified in assembly language notation with a suffix on the result register.

An instruction's type is usually the type of its result. For comparisons, however, the instruction's type is the type of its operands.

The result of a scalar comparison is a Boolean truth value (`0` or `1`) of type `L`, and the result of a vector comparison is a vector of Boolean truth values in a mask register.

In addition, some instructions reference 8-bit quantities (Bytes) and 16-bit quantities (Halfwords).

The convention used for numbering bits in a data type is that bit 2^n is referred to as bit n. In a range of bits within a data type from 2^n to 2^m where 2^n is a higher order bit than 2^m, the bits are specified as $n{:}m$.

2.1.1 Integer

The Word type `W` comprises 32-bit twos complement signed integers in the range

```
-2147483648 ... 2147483647
-0x80000000 ... 0x7fffffff
```

The Longword type L comprises 64-bit twos complement signed integers in the range

```
-9223372036854775808 ... 9223372036854775807
-0x8000000000000000  ... 0x7fffffffffffffff
```

Interpretations of these types as C `unsigned` integers mod 2^{32} and 2^{64} are supported in comparisons and implicitly in the `hi(`*j*`*`*k*`)` multiplication function. The assembly language syntax uses the notations `UW` and `UL` for `unsigned` comparisons.

Special instructions accelerate loads and stores of 8-bit and 16-bit integers from 64-bit Longword containers.

2.1.2 Floating-point

The Cray X1 system conforms to the IEEE-754 standard for floating-point arithmetic with the exception of subnormal numbers.

The 32-bit single precision floating-point type S has this format:

Bit	Definition
2^{31}	Sign
$2^{30}:2^{23}$	Exponent
$2^{32}:2^{0}$	Mantissa (implicit normalization)

The 64-bit double precision floating-point type D has this format:

Bit	Definition
2^{63}	Sign
$2^{62}:2^{52}$	Exponent
$2^{51}:2^{0}$	Mantissa (implicit normalization)

A 128-bit floating-point format is implemented by software. It comprises a standard 64-bit D value with 64 additional mantissa bits.

2.1.3 Addresses

Memory is addressed for instruction fetches, address and scalar register loads and stores, vector loads and stores, and atomic memory operations. All addresses have the same format and interpretation. They are 64-bit values that identify a byte offset in a virtual address space.

Addresses must be strongly aligned except for the special unaligned (`,ua`) load and store instructions. This means that instruction fetches and other 32-bit references must have zero bits in the lowest order two bit positions and that 64-bit references must have three lowest order zero bits.

The highest order bit positions (63:48) of a user virtual address must be zero. Distributed memory applications use bits 47:38 of the virtual address for a logical node number.

Data caches are always coherent, but the instruction caches are not. Self-modifying code must synchronize the instruction cache with an explicit `lsync,i` instruction.

The Cray X1 system addresses memory in big-endian order. If a 64-bit Longword is stored at aligned address *x*, a 32-bit load from *x*`+0` will retrieve its upper order 32 bits and a 32-bit load from *x*`+4` will retrieve its lower order 32 bits.

2.2 Registers

The Cray X1 processor has the following set of registers:

Table 1. Cray X1 Registers

`a0,...,a63`	64-bit address registers
`s0,...,s63`	64-bit scalar registers
`v0,...,v31`	32-bit or 64-bit vector registers
`vl`	Vector length register
`m0,...,m7`	Mask registers
`vc`	Vector carry register
`bmm`	64x64 bit matrix multiply register
`c0,...,c63`	Control registers

In addition to these registers, processor state information is contained in:

Table 2. Processor State Information

(no syntax)	Program counter
(no syntax)	Performance counters

In this manual, the term *register* refers to a specific register, such as `a23`, the term *register type* refers to the letter or letters that identify the specific set of registers, such as a for address registers, and *register designator* refers to the number that specifies a register in that set, such as `23`. CAL accepts either upper case or lower case for the register type, so `a23` and `A23` are treated the same.

2.2.1 Address and Scalar Registers

There are 64 address and 64 scalar registers, each 64 bits in width. They constitute computational way stations between memory and functional units in the processor for serial regions of the program. They also serve vectorized code with addresses, strides, and scalar values.

Registers `a0` and `s0` are unmodifiable zero values. Instructions that would write to them discard their results and may ignore exceptions.

Both the address and scalar registers are general-purpose and support the same memory reference instructions, immediate loads, integer functions, and conditional branches. However, address registers must be used for:

- Memory base addresses, offsets, and strides
- Indirect jump destinations and return addresses
- Vector element indexing
- Vector Length computation with `cvl()`
- Reading and writing `vl` and control registers
- Receiving the results of the mask analysis instructions `first()`, `last()`, and `pop()`
- Supplying the stride for a vector `scan()` and `cidx()`
- 8-bit and 16-bit accesses

And scalar registers must be used for:

- Scalar bit matrix multiplications
- Floating-point operations
- Scalar operands to vector operations

Unlike some other instruction set architectures, the Cray X1 system does not mandate a strict data typing rule. Integer operations may be performed on floating-point data and vice versa.

When 32-bit data enter a 64-bit address or scalar register, they are always right-justified and sign-filled, even if the value is notionally `unsigned`. 32-bit operations on 64-bit registers always ignore the upper bits and are guaranteed to return a sign-extended result.

There is a restriction that prohibits the use of both register designators *n* and *n*+32 or n and *n*-32 as operands for the same instruction if they are the same register types.

2.2.2 Vector Registers

There are 32 vector registers, each capable of holding 64 32-bit elements or 64 64-bit elements. They constitute computational way stations between the memory and the functional units of the processor for parallel regions of the program.

32-bit and 64-bit data are stored differently in vector registers. In particular, 32-bit data are not sign-extended as they are in the address and scalar registers.

A vector register holding the result of a 32-bit load or operation is defined for use only as an operand of a 32-bit store or operation. Similarly, a vector of 64-bit data can be used only in 64-bit instructions. Explicit conversion operations must be used to change the width of the data in the elements of vector registers.

This rule permits a hardware implementation to pack 32-bit vector elements in non-obvious ways that can vary from one generation to the next depending on pipe width. Packing permits 32-bit vector operations to execute at double speed.

When a vector register is written by a memory load or vector instruction, only the elements that are actually written are well-defined in the result vector register. The other elements, which may be those past the limiting value `vl` or which correspond to zero bits in the controlling mask register, become **undefined**. Programs must not assume that these other elements are preserved.

There is a restriction that prohibits the use of the same vector register as both the operand and the result of a type conversion operation.

The `vrip` instruction declares that the contents of the vector registers need no longer be maintained. It is used at the end of a vector sequence to avoid expensive context switch times.

2.2.3 Vector Length Register

The maximum number of elements that a vector register can hold is not actually specified by the architecture. It is only guaranteed to be a power of 2 and at least 64. It may vary between hardware implementations.

A vector's length is always a count of its elements, not its Bytes, Words, or Longwords. A vector of 32-bit data cannot hold any more elements than a vector of 64-bit data can.

The vector length register `vl` specifies the number of elements to be processed by vector register operations. Once set, it is an implicit operand to every vector register operation that follows.

Programs should use the `cvl()` function to compute legal and well-balanced Vector Length values.

Vector Length can be set to zero.

2.2.4 Mask Registers

Each of the eight mask registers contains a bit corresponding to each vector register element position. Since this may be larger than 64 bits, the instruction set contains instructions that manipulate mask registers directly.

Masks are set with the results of vector comparison operations. They can then be used to generate vector values with the `scan()` and `cidx()` functions. Masks are also used to control vector instructions on a per-element basis. Only the first four masks, `m0:m3`, can be used to control elemental vector operations. Values in `m4:m7` must be moved to `m0:m3` for use in vector instructions.

By software convention, mask register `m0` can be assumed to have every bit set.

2.2.5 Vector Carry Register

There is a single Vector Carry (vc) Register that is both an operand to and a result of the 64-bit vector `add with carry` and `subtract with borrow` instructions. Like the mask registers, it holds one bit for every vector register element position.

2.2.6 Bit Matrix Multiply Register

The Cray X1 system implements bit matrix multiplication (see Section 2.4.6, page 18). There is a 64 by 64 bit matrix multiply register that must be loaded from a vector register.

2.2.7 Control Registers

Access to most of the control registers is privileged to code running in kernel mode. These privileged control registers provide the means for programming the processor's address translation units and other critical functions.

Some control registers are available to user mode code and are therefore part of the instruction set architecture:

`c0`: Floating-point control (rounding mode and interrupt/exception masks)

`c1`: Floating-point status (interrupt/exception flags)

`c2`: Read-only MSP configuration information

`c3`: Read-only user time clock

`c4`: Read-only additional MSP information

`c28`: 32 performance counter enable flags

`c29`: 32 2-bit performance counter event selections

`c30`: Performance counter access control

`c31`: Performance counter value

2.2.8 Program Counter

The 64-bit virtual Program Counter holds the byte address of the next instruction to fetch. This counter is not visible to the user but its content is referenced in the description of some instructions, with the notation `pc`.

2.2.9 Performance Counters

The processor has 32 64-bit performance counters that are accessed indirectly through control register `c31`. Each counter can be programmed with one of four event codes that determine when it increments.

2.3 CAL Source Statement Format

The format of the Cray Assembly Language (CAL) source statement is briefly described here before we summarize the Cray X1 system machine instructions. For the complete list of machine instructions, see Chapter 7, page 103. Detail on the format of the listing files is shown in Figure 3, page 59.

Each machine instruction is denoted by a source statement with three fields separated by white space and optionally followed by a comment.

Label	*Result*	*Operand*	*Comment*

The *Label* field is optional for machine instructions. It can contain an identifier that must begin in column 1. The label is given the value of the instruction's byte address.

The *Result* field is always present. It typically contains the name of the instruction's result register and the type when necessary.

The *Operand* field is usually present. It typically contains a brief expression in C-like syntax with registers and constant expressions.

A *Comment* can follow the instruction. Comments begin with a semicolon, which is not part of the syntax for any instruction. A comment can also begin with an asterisk (*) in column 1.

Example:

```
Label    Result    Operand     Comment

top      a1        a2+a3       ; a1 gets the sum of a2 and a3
```

In the instruction descriptions that follow, similar instructions are often combined with a notation like:

```
ai,W or L    aj+ak or imm
```

instead of fully elaborating the four distinct instructions:

```
ai,W      aj+ak
ai,L      aj+ak
ai,W      aj+imm
ai,L      aj+imm
```

Operators in CAL statements very closely follow the syntax, meaning, and operator precedence rules of the C language. For details, see Section 6.9, page 94.

2.4 Scalar Instructions

This section gives an overview of the Cray X1 system scalar instruction set.

2.4.1 Immediate Loads

These CAL statements load the value of *expression* into the 64-bit a or s register.

```
ai      expression      ; Immediate load ai
si      expression      ; Immediate load si
```

However, because all Cray X1 system machine instructions are 32 bits wide, a 64-bit expression, depending on the bit pattern of the expression to be loaded, cannot necessarily be loaded into a register using a single machine instruction. To aid the programmer in loading a value of up to 64 bits into a register, the assembler supports these immediate load statements that the assembler expands into a sequence of one or more of the primitive 16-bit immediate load instructions described below.

The hardware supports a set of primitive immediate load instructions, each of which has a 16-bit expression (*expr* in the rest of this section) as part of the instruction and each of which modifies 1, 2, 3 or 4 16-bit quadrants (or Halfwords) of the result register.

For one set of primitives, one quadrant of the result register receives the 16-bit value (*expr*) specified in the machine instruction, and 0, 1, 2, or 3 of the quadrants containing the higher order bits may be filled with copies of the immediate value's uppermost bit for sign-extension; the remaining quadrants are set to zero. In other words, no part of the old value in the register remains unchanged.

For another set of primitives, one quadrant of the result register receives the 16-bit value (*expr*) specified in the machine instruction, and 0, 1, 2, or 3 of the quadrants containing the higher order bits may be filled with copies of the immediate value's uppermost bit for sign-extension; the remaining quadrants are left unchanged. In other words, a new value is merged with part of the old value in the register.

Which quadrants get modified and how they are modified depends on the quadrant (or Halfword) specifiers a (for bits 63:48), b (for bits 47:32), c (for bits 31:16) and d (for bits 15:0). The assembler uses two factors:

- The combination of specifiers (ab, abc, or abcd)
- The field in which the specifier is located (result field or operand field)

to determine which of the other three quadrants are modified and how.

The following table details the syntax and quadrant modifications.

Table 3. Quadrant Modifications

Result	Operand	a(63:48)	b(47:32)	c(31:16)	d(15:0)
a*i* or s*i*	*expr*:a	*expr*	0	0	0
a*i* or s*i*	*expr*:b	0	*expr*	0	0
a*i* or s*i*	*expr*:c	0	0	*expr*	0
a*i* or s*i*	*expr*:d	0	0	0	*expr*
a*i* or s*i*	*expr*:ab	*sign*	*expr*	0	0
a*i* or s*i*	*expr*:abc	*sign*	*sign*	*expr*	0
a*i* or s*i*	*expr*:abcd	*sign*	*sign*	*sign*	*expr*
a*i* or s*i*:a	*expr*	*expr*	-	-	-
a*i* or s*i*:b	*expr*	-	*expr*	-	-
a*i* or s*i*:c	*expr*	-	-	*expr*	-
a*i* or s*i*:d	*expr*	-	-	-	*expr*
a*i* or s*i*:ab	*expr*	*sign*	*expr*	-	-
a*i* or s*i*:abc	*expr*	*sign*	*sign*	*expr*	-

2.4.2 Scalar Memory References

Any of the data types W, L, S, or D can be used as *type* in the following instructions but only the size of *type* matters:

Table 4. Scalar Memory References

a*i* or s*i*,*type*	[a*j*+*expr*]	; Load from a*j* plus scaled offset *expr*
a*i* or s*i*,*type*	[a*j*+a*k*]	; Load from a*j* plus scaled index a*k*
a*i*	[a*j*],ua	; Load Longword containing unaligned address a*j*
[a*j*+*expr*]	a*i* or s*i*,*type*	; Store to a*j* plus scaled offset *expr*
[a*j*+a*k*]	a*i* or s*i*,*type*	; Store to a*j* plus scaled index a*k*

```
[aj]        ai,ua       ; Store to Longword containing unaligned address
                          aj
pref        [aj+expr]   ; Prefetch Longword from aj plus scaled offset expr
pref        [aj+ak]     ; Prefetch Longword from aj plus scaled index ak
```

Offset expressions (*expr*) and index values (a*k*) are always multiplied by the data size before being added to the a*j* base address. A 32-bit memory reference multiplies the offset by 4; a 64-bit reference multiplies the offset by 8. This means that alignment can be checked solely from the low order 2 or 3 bits of the base address a*j*. These bits are ignored for the unaligned (, ua) references and must be zero for the others.

Like all other instructions that enter 32-bit data into the 64-bit address and scalar registers, 32-bit loads sign-extend their results.

As a consequence of the Cray X1 system register numbering restriction mentioned above, indexed references must not use a*j* and a*k* registers whose register designators differ by exactly 32.

The constant offset values (*expr*) are signed twos complement 14-bit values prior to being scaled (`-8192...8191`). The byte offset ranges are therefore `-32768...32767` for 32-bit references and `-65536...65535` for 64-bit references.

The data cache prefetch (`pref`) instructions cannot raise any memory exceptions. Prefetches from bad addresses are therefore silently ignored.

2.4.3 Branches and Jumps

The Cray X1 system has six position-independent conditional branch instructions and one indirect jump instruction:

Table 5. Branches and Jumps

```
bz          ai or si,displ    ; Branch if == 0
bn          ai or si,displ    ; Branch if != 0
blt         ai or si,displ    ; Branch if < 0
ble         ai or si,displ    ; Branch if <= 0
bgt         ai or si,displ    ; Branch if > 0
```

bge	a*i* or s*i*,*displ*	; Branch if >= 0
j,a*i*	a*j*,SR or RT	; Jump to a*j* and set a*i* to pc+4 with optional subroutine call or return hint

The conditional branches test all 64 bits of a*i* or s*i*. 32-bit values will be correctly interpreted if the usual sign extension convention is observed.

The conditional branch instructions identify their targets with a signed twos complement 20-bit instruction displacement relative to the next instruction at pc+4. The offset range for conditional branches is therefore:

```
 -524287 ... 524288 instructions
-2097151 ... 2097152 bytes
```

relative to the branch instruction.

The single indirect jump instruction can be annotated with hints that distinguish subroutine calls and returns from other jumps. These hints are used by the hardware jump target prediction mechanism.

Jump destination addresses (a*j*) must be properly aligned so that their lowest two bits are zero.

2.4.4 Scalar Integer Functions

These are the integer operations available to address and scalar registers. To avoid clutter, only the address register forms are shown here. The scalar register forms are analogous, except for the lack of the `cvl()` function. Type `L` is the default type. In many instructions, the a*k* register can be replaced with an unsigned 8-bit immediate constant value (*imm*).

Table 6. Scalar Integer Functions

a*i*,W or L	a*j*+a*k* or *imm*	; Addition
a*i*,W or L	a*j*-a*k* or *imm*	; Subtraction
a*i*,W or L	a*j**a*k*	; Multiplication
a*i*,W or L	a*k*/a*j*	; Signed division
a*i*	hi(a*j**a*k*)	; Upper 64 bits of full 128-bit multiplication
a*i*	a*j*&a*k* or *imm*	; Logical AND

a*i*	a*j*|a*k* or *imm*	; Logical OR
a*i*	a*j*^a*k* or *imm*	; Logical XOR
a*i*	~a*j*&a*k*	; Logical AND with complement
a*i*	~a*j*^a*k*	; Logical Equivalence
a*i*,W or L	a*j*<<a*k* or *imm*	; Left shift
a*i*, W or L	a*j*>>a*k* or *imm*	; Logical right shift
a*i*	+a*j*>>a*k* or *imm*	; Arithmetic right shift
a*i*,L or UL	a*j*<a*k* or *imm*	; Comparison
a*i*,W or L	lz(a*j*)	; Leading zero count
a*i*,W or L	pop(a*j*)	; Population count
a*i*	a*j*?a*k*:a*i*	; Conditional move, a*i* gets a*k* if a*j* != 0
a*i*	a*j*?a*i*:a*k*	; Conditional move, a*i* gets a*k* if a*j* == 0
a*i*	cvl(a*k* or *imm*)	; Compute Vector Length

The rules and guidelines below apply to integer functions:

- A properly encoded integer function can raise no exception.
- Division by zero is undefined, as is the signed division of the most negative twos complement integer by -1:

  ```
  -9223372036854775808 / -1
  ```

  The result cannot be represented in 64 bits.
- Shift count values are interpreted as unsigned integers of the same size as the shifted value. This means that the upper 32 bits of the shift count are ignored for 32-bit shifts, and that negative shift counts are out of range.
- Shift count values that are out of range yield zero results for logical left and right shifts and a word full of sign bits for arithmetic right shifts.
- The hi(a*j**a*k*) function is useful for multiprecision integer arithmetic and for implementing division by constant values that are not exact powers of 2. Compute a normalized fixed-point reciprocal of the denominator accurate to 64 bits at compilation time, then implement division with hi(a*j**a*k*) and a right shift.

- The Vector Length computation function `cvl()` computes a balanced and legal Vector Length value for a given count of remaining loop iterations. Specifically,

Value of cvl(a*k*)	If
0	a*k* < 0
a*k*	0 <= a*k* <= MAXVL
a*k*/2	MAXVL<a*k* < 2 × MAXVL
MAXVL	a*k* > 2 × MAXVL

  where MAXVL is the actual maximum vector length at run time.

- As long as 32-bit data are maintained in the canonical sign extended format in the 64-bit address and scalar registers, an explicit integer type conversion is necessary only when one is:
  - converting unsigned 32-bit integer data to a 64-bit integer type, or
  - converting a 64-bit integer type to a 32-bit integer type

  The first case is best implemented with an immediate load of zero into the upper 32 bits of the register (a*i*:ab 0). The second case is best implemented with a 32-bit integer addition to zero (a*i*,W a*j*+0). And this latter conversion is not required when the result is used only as a operand to 32-bit operations.

- Here are two useful idioms:

```
ai,UL     aj< 1      ; C logical negation (ai !aj)
ai,UL     a0<ak      ; C logical normalization (ai !!ak)
```

- Only a less-than comparison function is necessary. It yields 1 for true and 0 for false. The full set of integer relations can be constructed thus:

Table 7. Integer Relations

a*i*	a*j*<a*k*	; a*j* < a*k*
a*i*	a*k*<a*j*	
a*i*,UL	a*i*<1	; a*j* <= a*k*
a*i*	a*k*<a*j*	; a*j* > a*k*
a*i*	a*j*<a*k*	
a*i*,UL	a*i*<1	; a*j* >= a*k*

- When a comparison is used as the predicate of a conditional branch, the second instruction can be removed by reversing the branch condition.

2.4.5 Scalar Floating-point Functions

Unlike the integer functions, scalar floating-point instructions are available only to the scalar (*sj*) registers. There is no default data type, nor are there immediate operand forms.

Table 8. Scalar Floating-point Functions

si,S or D	*sj*+*sk*	; Addition
si,S or D	*sj*-*k*	; Subtraction
si,S or D	*sj***sk*	; Multiplication
si,S or D	*sk*/*sj*	; Division
si,S or D	abs(*sj*)	; Arithmetic absolute value
si,S or D	sqrt(*sj*)	; Square root
si,S or D	cpys(*sj*,*sk*)	; Copy value of *sj* with sign of *sk*
si,S or D	*sj*==*sk*	; Equality comparison (unordered yields 0)
si,S or D	*sj*!=*sk*	; Inequality comparison (unordered yields 1)
si, S or D	*sj*<*sk*	; Less than comparison (unordered yields trap)
si,S or D	*sj*<=*sk*	; Less than or equal comparison (unordered yields trap)
si,S or D	*sj*?*sk*	; Test ordered (ordered yields 1, unordered yields 0)
si,*t1*	*sj*,*t2*	; Convert from type *t2* to distinct type *t1* (not both integer)
si,W or L	round(*sj*),S or D	; Convert floating-point to integer, rounding to nearest
si,W or L	trunc(*sj*),S or D	; Convert floating-point to integer, rounding to zero
si,W or L	ceil(*sj*),S or D	; Convert floating-point to integer, rounding up
si,W or L	floor(*sj*),S or D	; Convert floating-point to integer, rounding down

The rules and guidelines below apply to floating-point functions:

- Comparisons yield the 64-bit integer values 0 and 1 as results.

- To ensure that IEEE-754 signed zero values are properly recognized, programs should use a floating-point comparison against s0 before a conditional branch instead of using bz and bn directly on floating-point data.
- IEEE-754 defines the absolute value function as a non-arithmetic operation, or one that should not trap on an invalid operand. Implement a non-arithmetic absolute value function with cpys(s*j*,s0).
- Similarly, a non-arithmetic negation should also be implemented with cpys() from a logical complement of the operand.
- Arithmetic floating-point operations signal floating-point exceptions and cause traps depending on the mask fields in the floating-point control register c0. Floating-point traps are not precise. The lsync fp instruction should be used when it is necessary to force pending traps to be taken before proceeding.

2.4.6 Bit Matrix Multiplication

There are three bit matrix multiply instructions:

Table 9. Bit Matrix Multiply Instructions

bmm	v*k*	; Bit matrix load
s*i*	bmm(s*k*)	; Scalar bit matrix multiply
v*i*	bmm(v*k*),m*m*	; Vector bit matrix multiply

The bmm v*k* instruction loads bits from the vector register v*k* into the bmm register.

The s*i* bmm(s*k*) instruction performs the basic bit matrix multiplication operation. Each bit *j* of the 64-bit integer result s*i*, counting from the highest order bit position down to the lowest, is computed thus:

```
si[j] = pop(sk & bmm[j]) (mod 2)
```

where bmm*j* is the *j* the row of the bmm register.

In other words, bit *j* of the result s*i* is set if the logical AND of s*k* and element *j* of the vector register that was loaded into the bmm register has an odd number of bits set. When the bmm register is loaded such that exactly one bit is set in each row and each column, it performs a bit permutation function such as a centrifuge.

The bit matrix multiplication can also be viewed as the matrix product (mod 2) of a 1-by-64 row vector in *sk* and the 64-by-64 transpose of bmm. Within this perspective, each 64-bit word loaded into bmm constitutes a column.

The *vi* bmm(*vk*),*mm* instruction is described in the Vector instructions section (Section 2.5, page 22).

2.4.7 Byte and Halfword Access

The Cray X1 system instruction set supports direct memory references to 32-bit and 64-bit data. The functions described in this section accelerate access to 8-bit (Byte) and 16-bit (Halfword) data. They manipulate only address register operands.

Table 10. Byte and Halfword Access

a*i*	extb(a*j*,a*k*)	; Extract unsigned Byte from a*j* at a*k*
a*i*	exth(a*j*,a*k*)	; Extract unsigned Halfword from a*j* at a*k*
a*i*	mskb(a*j*,a*k*)	; Clear Byte in a*j* at a*k*
a*i*	mskh(a*j*,a*k*)	; Clear Halfword in a*j* at a*k*
a*i*	insb(a*j*,a*k*)	; Shift Byte a*j* into position a*k*
a*i*	insh(a*j*,a*k*)	; Shift Halfword a*j* into position a*k*

All of these functions use the lowest order 2 or 3 bits of a*k* to identify a 16-bit Halfword or 8-bit Byte or position in a 64-bit field in big-endian order. The 64-bit field is assumed to hold the Longword containing the byte addressed by a*k*.

The functions listed above perform the following operations:

- The ext() functions extract the indexed field, right-justified and zero-extended to 64 bits.
- The msk() functions clear the indexed field.
- The ins() functions truncate the value of a*j* and shift it into the indexed field.

To load an unsigned Byte from memory address a2, use:

```
a1      [a2],ua
a1      extb(a1,a2)
```

To store a Byte in a1 to memory address a2, use:

```
a3      [a2],ua
a3      mskb(a3,a2)
a4      insb(a1,a2)
a3      a3|a4
[a2]    a3,ua
```

Note: This store sequence is not atomic with respect to stores into the same 64-bit Longword from other processors. Use the atomic aax operation (Section 2.4.8, page 20) when such conflicts are possible.

The 16-bit Halfword access functions are defined only for properly aligned (even) a*k* addresses.

2.4.8 Scalar Atomic Memory Operations

There are three basic atomic memory operations. Two of them have variant forms to be used when no result is necessary. The result of an atomic memory operation is always the old Longword value from memory.

Table 11. Scalar Atomic Memory Operations

[a*j*]	a*i*,aadd	; Add a*i* to Longword addressed by a*j*
a*i*	[a*j*],afadd,a*k*	; Atomic add, returning old value
[a*j*]	a*i*,aax,a*k*	; AND a*i* to Longword addressed by a*j*, then XOR with a*k*
a*i*	[a*j*],afax,a*k*	; Atomic logical, returning old value
a*i*	[a*j*],acswap,a*k*	; Store a*k* to Longword addressed by a*j* if its old value equals a*i*

Notes on atomic scalar memory operations:

- Only atomic operations on aligned 64-bit Longwords are provided. Analogous atomic operations on shorter fields are synthesized in software.
- Register a*i* is used as an operand to afax and acswap and, as a result, is also overwritten with the old value from memory.

- The general atomic logical operation can be used to synthesize more specific useful operations:

	ai value	*ak* value
AND *x*	*x*	0
OR *x*	*x* complement	*x*
XOR *x*	0 complement	*x*
Equivalence *x*	0 complement	*x* complement
Set bits *x* and clear *y*	(x OR y) complement	x
Mask with *x* and store *y*	*x*	*y*

- Atomic memory operations can be suffixed with `,NA` to hint to the caches that the Longword should not be allocated, or with `,EX` to hint that the Longword should be exclusively allocated. The default hint is `,NA`. Cache hints are ignored on atomic memory operations.
- Atomic memory operations are not ordered with respect to other memory references in the same processor, apart from those required by register-to-register data dependences. They must be surrounded by `gsync` instructions to ensure proper ordering.

2.4.9 Other Scalar Instructions

These nonprivileged scalar instructions do not fit neatly into any of the categories above, and so they are described here for completeness:

Table 12. Other Scalar Instructions

`syscall`	*expr*	; System call
`break`	*expr*	; Debugging breakpoint
a*i*,W or L	s*k*	; Transfer s*k* to a*i*
s*i*,W or L	a*k*	; Transfer a*k* to s*i*
a*i*	c*k*	; Read Control register
c*i*	a*k*	; Write Control register

2.5 Vector Instructions

This section gives an overview of the Cray X1 system vector instruction set.

2.5.1 Elemental Vector Operations

The Cray X1 system vector instruction set contains vector versions of most scalar integer functions, floating-point functions, and memory references. These elemental vector operations process each element independently. They execute under control of a mask register (`m0, . . . ,m3`) and the vector length register (`vl`). Though there are 8 mask registers, only the first 4 can be used in the vector instructions.

The assembler's default controlling mask register is `m0`. By software convention, every bit in `m0` is always set.

In the vector result of an elemental vector operation, those elements that are past `vl` or which correspond to a zero bit in the controlling mask register become undefined (merges are the exception).

Elemental vector operations therefore have these semantics:

```
for (I = 0; I < MAXVL; I++)
    if (I >= vl || !mm[I])
        vi[I] = undefined;
    else
        vi[I] = vj[I] op (sk or vk[I]);
```

2.5.2 Vector Memory References

The Cray X1 system loads and stores vector registers from a sequence of properly aligned byte addresses. The address sequence is computed from an aligned base address a*j* and either a scalar stride value or a vector of 64-bit offset values. Strided references add multiples of the stride value to the base address, while gather/scatter references add the corresponding offset vector element value. Both scalar strides and vector offset values are scaled by the data size.

Table 13. Vector Memory References

v*i*, *type*	[a*j*, a*k* or *imm*] , m*m*	; Strided load
v*i*, *type*	[a*j*, v*k*] , m*m*	; Gather
[a*j*, a*k* or *imm*]	v*i*,*type*,m*m*	; Strided store

[a*j*, v*k*]	v*i*, *type*, m*m*	; Scatter with distinct offsets
[a*j*, v*k*]	v*i*, *type*, m*m* , ord	; Scatter with arbitrary offsets

The rules and guidelines below apply to vector memory references:

- Immediate stride values are 6-bit signed twos complement integers in the range -32...31
- A poorly aligned a*j* base address may or may not cause a trap if vl is zero or the controlling mask m*m* is clear.
- All vector memory reference instructions can be annotated with an optional cache hint code:
  - , NA Data should not allocate space in cache if not present
  - , SH New allocations should be in shared state
  - , EX New allocations should be in exclusive state (default)

2.5.3 Elemental Vector Functions

Most elemental vector functions with two operands permit the use of s*k* as a fixed scalar operand in place of v*k*.

Table 14. Elemental Vector Functions

v*i*, *type*	v*j*+s*k* or v*k*, m*m*	; Addition
v*i*, *type*	s*k* or v*k*-v*j*, m*m*	; Subtraction
v*i*, vc	v*j*+s*k* or v*k*, m*m*	; 64-bit addition with carry
v*i*, vc	s*k* or v*k*-v*j*, m*m*	; 64-bit subtraction with borrow
v*i*, *type*	v*j**s*k* or v*k*, m*m*	; Multiplication
v*i*	hi (v*j**s*k* or v*k*) , m*m*	; Upper 64 bits of full 128-bit multiplication
v*i*, S or D	s*k* or v*k*/v*k*, m*m*	; Floating-point division
v*i*, W or L	v*j*&s*k* or v*k*, m*m*	; Logical AND
v*i*, W or L	v*j*| s*k* or v*k*, m*m*	; Logical OR
v*i*, W or L	v*j*^s*k* or v*k*, m*m*	; Logical XOR
v*i*, W or L	~v*j*&s*k* or v*k*, m*m*	; Logical AND with complement

vi, W or L	~vj^sk or vk, mm	; Logical Equivalence
vi, W or L	vj<<sk or vk, mm	; Logical left shift
vi, W or L	vj>>sk or vk, mm	; Logical right shift
vi, W or L	+vj>>sk or vk, mm	; Arithmetic right shift
vi, W or L	lz (vj) , mm	; Leading zero count
vi, W or L	pop (vj) , mm	; Population count
vi	bmm (vk) , mm	; Bit matrix multiplication
vi, S or D	abs (vj) , mm	; Arithmetic absolute value
vi, S or D	sqrt (vj) , mm	; Square root
vi, S or D	cpys (vj, vk) , mm	; Copy of vj with signs of vk
vi, t1	vj, t2 , mm	; Convert type t2 to distinct type t1
vi, W or L	round (vj) , S or D, mm	; Convert floating-point to integer, rounding to nearest
vi, W or L	trunc (vj) , S or D, mm	; Convert floating-point to integer, rounding to zero
vi, W or L	ceil (vj) , S or D, mm	; Convert floating-point to integer, rounding up
vi, W or L	floor (vj) , S or D, mm	; Convert floating-point to integer, rounding down
vi, W or L	mm?sk or vk:vj	; Merge
mi, type	vj==sk or vk, mm	; Compare for equal
mi, type	vj!=sk or vk, mm	; Compare for not equal
mi, type	vj<sk or vk,mm	; Compare for less than
mi, type	vj<=sk or vk, mm	; Compare for less than or equal
mi, type	vj>sk, mm	; Compare for greater than
mi, type	vj>=sk or vk,mm	; Compare for greater than or equal
mi, S or D	vj?sk or vk, mm	; Compare for unordered

The rules and guidelines below apply to elemental vector functions:

- The comparisons <, <=, >, and >= can also use types UW and UL for unsigned integer comparisons.
- There is no vector form of integer division.
- There is no vector/scalar division.

- As in scalar floating-point functions, non-arithmetic absolute value and negation should be implemented with `cpys()`.
- Subtraction with borrow will clear bits in vc when a borrow occurs.
- Unselected elements of vc are cleared in an add-with-carry or subtract-with-borrow function.
- Vectors of 32-bit data must not be used as operands to 64-bit functions and vice versa.
- Comparisons are elemental operations, but every bit of the resulting mask m*i* is well defined. Result mask bits that correspond to element positions past vl or to zero bits in the controlling mask m*m* are set to zero. The result mask m*i* is therefore a subset of the controlling mask m*m*.
- Most commutative functions with a v*j* *op* s*k* syntax have a reversed s*k* *op* v*j* synonym supported by the assembler. For details, see Section 2.8, page 31.
- The `bmm()` function is the elemental vector analog to the scalar s*i* `bmm(`s*k*`)` instruction.

2.5.4 Mask Operations

The Cray X1 system provides operations that operate directly on mask registers. This feature improves decoupled execution and permits the maximum vector length to exceed the width of the Address and scalar registers.

Table 15. Mask Operations

m*i*	m*j*&m*k*	; Logical AND
m*i*	m*j*|m*k*	; Logical OR
m*i*	m*j*^m*k*	; Logical XOR
m*i*	~m*j*&m*k*	; Logical AND with complement
m*i*	~m*j*^m*k*	; Logical equivalence
m*i*	`fill(`a*k*`)`	; Set leading a*k* bits, clear remainder
a*i*	`first(`m*k*`)`	; Find lowest set bit index (MAXVL if none)
a*i*	`last(`m*k*`)`	; Find highest set bit index (MAXVL if none)
a*i*	`pop(`m*k*`)`	; Count number of bits set

v*i*	scan(a*k*, m*j*)	; Running sums of mask bits (times a*k*)
v*i*	cidx(a*k*, m*j*)	; Indices of set mask bits (times a*k*)
v*i*	cmprss(v*k*, m*j*)	; Compress vector

The rules and guidelines below apply to mask operations:

- Any of the mask registers can be used in the mask operation instructions, but only m0–m3 can be used in the vector instructions.
- The scan(), cidx(), and cmprss() functions use vl. The other mask operations manipulate all MAXVL bits of the masks.
- The scan() function generates a vector from a mask register, a stride, and the vector length register. For each element position 0 <= n < vl,

  $vi[n] = ak \times \Sigma_{0 \le t < n} mj[t]$

  The remaining elements of the result are **undefined.**

  Equivalently, scan() can be defined with this algorithm:

```
for (I = sum = 0; I < MAXVL; I++)
    if (I >= vl)
        vi[I] = undefined;
    else {
        vi[I] = sum;
        if (mj[I])
            sum += ak;
    }
```

  The scan() function is particularly useful for vectorizing conditionally incremented loop induction variables and for constructing gather/scatter offset vectors for compression and expansion.

- The cidx() function defines the leading elements of the result vector register v*i* and undefines the remainder. The number of elements written to v*i* equals the number of bits set in the mask m*j*.

```
for (I = sum = 0; I < vl; I++)
    if (mj[I])
        vi[sum++] = ak * I;
for (; sum < MAXVL; sum++)
    vi[sum] = undefined;
```

- The `cmprss()` function is similar to `cidx()` except that it stores values from another vector register.

```
for (I = sum = 0; I < vl; I++)
    if (mj[I])
        vi[sum++] = vk[I];
for (; sum < MAXVL; sum++)
    vi[sum] = undefined;
```

2.5.5 Other Vector Instructions

These vector instructions do not fit neatly into any of the categories above, so they are described here for completeness.

Table 16. Other Vector Instructions

`vl`	a*k*	; Set Vector Length
a*i*	`vl`	; Retrieve Vector Length
a*i* or s*i*,`W` or `L`	v*k*, a*j* or *imm*	; Read element
v*i*, a*j* or *imm*	a*k* or s*k*,`W` or `L`	; Write element
`bmm`	v*k*	; Load bit matrix
`vrip`		; Declare vector state dead

The rules and guidelines below apply to these vector instructions:

- Immediate vector element indices (*imm*) are unsigned 6-bit integer values, so only the first 64 elements of a vector can be directly indexed.
- Vector element indices (a*j*) that are outside the range `0 <=` a*j* `<MAXVL` are undefined and may cause a trap.
- Writing a value outside the range `0 <=` a*k* `<= MAXVL` to `vl` will cause a trap. Use the `cvl()` function to compute balanced and legal vector lengths before writing `vl`.
- The `bmm` v*k* instruction copies the leading 64-bit elements of v*k* into the corresponding rows of the 64 by 64 bit matrix. Only `min(vl,64)` elements are written. The remaining rows are cleared, so every bit in the matrix becomes well defined.

This operation is not elemental because it does not operate under control of a mask register. The v*k* register must therefore be completely defined up to `vl` with 64-bit values.

- The `vrip` instruction undefines all of the vector registers, the vector carry register `vc`, and the mask registers, apart from `m0`. If the contents of the `bmm` register are defined, they remain defined. Mask register `m0` remains defined if it still has all of its bits set; otherwise, it too becomes undefined.

  The `vrip` instruction is used at the end of a sequence of vector instructions to reduce the size of the processor state that must be saved and restored when switching processor contexts, and also to release physical registers in implementations that rename the vector registers.

a*i*	m*k*, a*j* or *imm*	; Read mask bits
m*i*, a*j* or *imm*	a*k*	; Write mask bits

The a*j* or immediate value is multiplied by 64 to index the first mask bit to be transferred to or from the highest-order bit position in the address register. If it is out of range, the results are undefined and a trap may ensue.

2.6 Memory Ordering

The Cray X1 system provides minimal implicit memory ordering guarantees between different references from a processor to the same word of memory. Scalar references are mutually well ordered, but scalar/vector and vector/vector interactions generally are not. While atomic memory operations on the same Longword are mutually ordered, they are not ordered relative to scalar and vector references.

Explicit memory ordering instructions must appear in the program to ensure that references to overlapping words will execute in the right order.

Suppose that *A* and *B* are different references to memory from a given processor. They can be scalar loads, scalar stores, atomic memory operations, or distinct elements of vector load or store instructions. Suppose that *A* precedes *B* in the naive program order, and that they reference overlapping words of memory. If both *A* and *B* are loads, no ordering question pertains. Otherwise, the Cray X1 system guarantees that *B* will reference memory after *A* only if:

- *A* and *B* are both scalar references

- *B* is a store whose address or data cannot be computed without the result of the load *A*
- *A* and *B* are elements of the same ordered vector scatter or zero-stride vector store
- *A* and *B* are elements of the same unordered vector scatter and are storing the same value

All other intraprocessor ordering cases require the presence of explicit memory ordering instructions.

Table 17. Explicit Memory Ordering Instructions

`gsync`	`aj`	; Global ordering of all prior references before all later references
`gsync`	`aj,cpu`	; Global ordering, with all prior instructions complete
`gsync`	`aj,a`	; Global ordering of prior scalar loads before all later references
`gsync`	`aj,r`	; Global ordering of all prior references before later scalar stores
`msync`	`aj`	; MSP ordering of all prior references before all later references
`msync`	`aj,p`	; MSP ordering of all prior stores before all later loads
`msync`	`aj,v`	; MSP ordering of all prior vector references before all later vector references
`lsync`	`s,v`	; Local ordering of prior scalar references before later vector references
`lsync`	`v,s`	; Local ordering of prior vector references before later scalar references
`lsync`	`vr,s`	; Local ordering of prior vector loads before later scalar references
`lsync`	`v,v`	; Local ordering of prior vector references before later vector references
`lsync`	`vj,v`	; Local ordering of latest v*j* reference before later vector references
`lsync`	`vj,v,el`	; Local elemental ordering of latest v*j* reference before later vector references
`lsync`	`fp`	; Local ordering of prior floating-point traps before later instructions
`lsync`	`i`	; Local ordering of prior stores to instruction memory before later instructions

Both the gsync and msync instructions require an a*j* processor mask operand that specifies a set of processors within the local MSP group of four. They perform a local barrier operation that must be satisfied by all processors in the a*j* mask with identical mask values before execution can proceed. The a*j* processor mask is right-justified and the lowest-order bit position corresponds to processor 0 in the local group.

The conjunction of the barrier semantics with the memory ordering effects is unfortunate. When no barrier is needed, a processor mask must be used that has just the bit set that corresponds to the issuing processor.

Interprocessor memory reference ordering is also weakly guaranteed. The architecture specifies only that the references of one processor will be seen in the same order by others, and that once a store is visible to another processor, it will be visible to all. (References to I/O space in system code by a single processor are implicitly mutually ordered.)

2.7 Summary of Rules

To summarize, these rules apply to the Cray X1 system instruction set:

- When a 32-bit datum enters an address or scalar register, it is always right-justified and sign extended to 64 bits.
- When an address or scalar register is used as a 32-bit operand, its upper 32 bits are ignored.
- On memory reference instructions, all immediate offsets, a*k* indices, and v*k* gather/scatter offset values are scaled by the data size.
- When *j* and *k* are distinct register operands of the same kind to the same instruction, then *j* != *k* (mod 32).
- A vector register cannot be both operand and result of the same vector type conversion instruction.
- 32-bit vector results must not be used as operands to 64-bit operations and vice versa.
- Memory addresses must be strongly aligned, apart from , ua references.
- Result vector register elements that are not written become undefined and cannot be assumed to be preserved.

- Explicit memory ordering instructions are necessary for proper ordering of overlapping references that are not both scalar and are not both loads, unless there is an implicit ordering due to register data dependences.
- gsync instructions are required around atomic memory operations.
- MAXVL is a power of 2 and at least 64 but cannot be assumed to be fixed.

2.8 Special Syntax Forms

For programmer convenience, the assembler supports the following special syntax forms for machine instructions:

Table 18. Special Syntax Forms

Special Form		Translated Into	
a*i*	a*j*	a*i*,L	a*j*|0
s*i*	s*j*	s*i*,L	s*j*|0
v*i*	v*j*	v*i*,L	v*j*|s0,m0
m*i*	m*j*	m*i*	m*j*|m*j*
a*i*	*expr*	a*i*	*expr1*:a[1]
		a*i*:b	*expr2*
		a*i*:c	*expr3*
		a*i*:d	*expr4*
a*i*	*symbol*	a*i*	*expr1*:a[2]
		a*i*:b	*expr2*
		a*i*:c	*expr3*
		a*i*:d	*expr4*
a.SP		a63	
a.FP		a62	
a.EA		a61	

1 The assembler analyzes *expr* and tries to generate the minimum number of load immediate instructions.

2 If the symbol is relocatable, the assembler always generates four instructions and the loader fills in the values.

a.RA	a60
a.CI	a59
a.CSP	a58
a.CFP	a57

CAL Pseudo Instruction Overview [3]

Pseudo instructions direct the assembler in its task of interpreting source statements and generating an object file. This chapter briefly describes all the pseudo instructions. For detailed descriptions of the pseudo instructions, see Chapter 8, page 129. Chapter 9, page 213 further describes the use of the pseudo instructions that are used for defined sequences; that is, for defining macros and opdefs.

Pseudo instructions are classified and described according to their applications, as follows:

Class	Pseudo instructions
Program control	IDENT, END, COMMENT
Loader linkage	ENTRY, EXT, START, OSINFO
Mode control	BASE, QUAL, EDIT, FORMAT
Section control	SECTION, STACK, ORG, LOC, BITW, BSS, ALIGN
Message control	ERROR, ERRIF, MLEVEL, DMSG, MSG, NOMSG
Listing control	LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, ENDTEXT
Symbol definition	=, SET, MICSIZE, DBSM
Data definition	CON, BSSZ, DATA, VWD
Conditional assembly	IFA, IFC, IFE, IFM, SKIP, ENDIF, ELSE
Micro definition	CMICRO, MICRO, OCTMIC, DECMIC, HEXMIC
File control	INCLUDE
Defined sequences	MACRO, OPDEF, DUP, ECHO, ENDM, ENDDUP, STOPDUP, LOCAL, OPSYN, EXITM, NEXTDUP

Note: You can specify pseudo instructions in uppercase or lowercase, but not in mixed case.

3.1 Pseudo Instruction Format

The source statement format for the pseudo instructions has the same format as used for symbolic machine instructions, but the contents of the fields differ.

Label *Result* *Operand* *Comment*

label *name* *arguments* *; comment*

The fields are separated by white space. Depending on the pseudo instruction, the *label* field may be required, optional, or ignored. The *name* in the Result field is the name of the pseudo instruction. The *name* may be all uppercase or all lowercase but not mixed case. No other subfield besides the *name* is in the Result field. The *arguments* in the Operand field may be required, optional, or ignored. Comments are optional.

3.2 Program Control

The program control pseudo instructions define the limits of a program module and are as follows:

Pseudo	Description
IDENT	Marks the beginning of a program module.
END	Marks the end of a program module.
COMMENT	Enters comment (for example, a copyright) into the generated binary load module.

3.3 Loader Linkage

The loader linkage pseudo instructions provide for the linking of multiple object program modules into one executable program:

Pseudo	Description
ENTRY	Specifies symbols defined as objects, function entry points, or values so that they can be used by other program modules linked by a loader.
EXT	Specifies linkage to objects, function entry points, or values defined as entry symbols in other program modules.
START	Specifies symbolic address at which execution begins.

OSINFO	Specifies information to be passed to the loader about how the object file is to be linked.

3.4 Mode Control

Mode control pseudo instructions define the characteristics of an assembly:

Pseudo	Description
BASE	Specifies data as being octal, decimal, hex, or a mixture of octal and decimal.
QUAL	Designates a sequence of code where symbols may be defined with a qualifier, such as a common routine with its own labels.
EDIT	Turns editing on or off.
FORMAT	Changes the format to old or new.

3.5 Section Control

Section control pseudo instructions control the use of sections and counters in a CAL program:

Pseudo	Description
SECTION	Defines specific program sections.
STACK	Increments the size of the stack.
ORG	Resets location and origin counters.
LOC	Resets location counter.
BITW	Sets the current bit position relative to the current Longword.
BSS	Reserves memory. The reserved memory is uninitialized.
ALIGN	(Deferred implementation) Aligns code.

3.6 Message Control

The message control pseudo instructions are as follows:

Pseudo	Description
ERROR	Sets an assembly error flag
ERRIF	Sets an assembly error flag according to the conditions being tested

MSG and NOMSG	Specifies that certain messages should be issued or not issued
DMSG	Issues a comment-level message containing the string found in the operand field

3.7 Listing Control

The listing control pseudo instructions are as follows:

Pseudo	Description
LIST	Controls listing by specifying particular listing features that will be enabled or disabled
SPACE	Inserts blank lines in listing
EJECT	Begins new page
TITLE	Prints main title on each page of listing
SUBTITLE	Prints subtitle on each page of listing
TEXT	Declares beginning of global text source
ENDTEXT	Terminates global text source

3.8 Symbol Definition

The symbol definition pseudo instructions are as follows:

Pseudo	Description
=	Equates a symbol to a value; not redefinable.
SET	Sets a symbol to a value; redefinable.
MICSIZE	Equates a symbol to a value equal to the number of characters in micro string; redefinable.
DBSM	Generates a named label entry in the debug symbol tables with a specific type specified.

3.9 Data Definition

Data definition pseudo instructions are the only pseudo instructions that generate object binary. The only other instructions that are translated into object binary are the symbolic machine instructions. An instruction that generates

binary cannot be used within a section that does not allow instructions, data, or both.

The data definition pseudo instructions are as follows:

Pseudo	Description
CON	Places an expression value into one or more Longwords
BSSZ	Generates bytes that have been initialized to 0
DATA	Generates one or more Longwords of numeric or character data
VWD	Generates a variable-width field, 0-64 bits wide, of data

3.10 Conditional Assembly

The conditional assembly pseudo instructions permit the optional assembly or skipping of source code.

The conditional assembly pseudo instructions are as follows:

Pseudo	Description
IFA	Tests expression attributes; address and relative attributes.
IFE	Tests two expressions for some assembly condition; less than, greater than, and equal to.
IFC	Tests two character strings for assembly condition; less than, greater than, and equal to.
IFM	Test for machine characteristics.
SKIP	Unconditionally skip subsequent statements.
ENDIF	Terminates conditional code sequence.
ELSE	Reverses assembly condition.

3.11 Micro Definition

Through the use of micros, programmers can assign a name to a character string and subsequently refer to the character string by its name. A reference to a micro results in the character string being substituted for the name before assembly of the source statement containing the reference.

The following pseudo instructions specify micro definition:

Pseudo	Description
CMICRO	Constant , non-redefinable micro; assigns a name to a character string.
MICRO	Redefinable micro; assigns a name to a character string.
OCTMIC	Converts the octal value of an expression to a character string and assigns it a redefinable name.
DECMIC	Converts the decimal value of an expression to a character string and assigns it a redefinable micro name.
HEXMIC	Converts the hex value of an expression to a character string and assigns it a redefinable micro name.

In addition to the micros previously listed, the assembler provides predefined micros. They can be specified in all uppercase or all lowercase, but not mixed case. CAL provides the following predefined micros:

Micro	Description
$DATE	Current date: *' mm / dd / yy '*
$JDATE	Julian date: *' yyddd '*
$TIME	Time of day - *' hh : mm : ss '*
$MIC	Micro character: double quotation mark (")
$CNC	Concatenation character: underscore (_).
$QUAL	Name of qualifier in effect; if none, the string is null.
$CPU	Target machine: 'CRAY X1'.
$CMNT	Comment character used with the new format: semicolon (;).
$APP	Append character used with the new format: backslash (must be immediately followed by a newline).

The following example illustrates the use of a predefined micro ($DATE):

```
DATA  'THE DATE IS "$DATE"'
DATA  'THE DATE IS 01/01/00'          <edited line>
```

You can reference micro definitions anywhere in a source statement, except in a comment, by enclosing the micro name in quotation marks. If column 72 of a line is exceeded because of a micro substitution, the assembler creates additional continuation lines. No replacement occurs if the micro name is unknown or if one of the micro marks was omitted.

In the following example, a micro called PFX is defined as the character string ID. A reference to PFX is in the label field of a line.

```
"PFX"TAG  A1    S1      ; Left-shifted three spaces when edited.
```

In the following example, before the line is interpreted, the assembler substitutes the definition for PFX producing the following line:

```
IDTAG   A1      S1      ; Left-shifted three spaces when edited.
```

The following example shows the use of the predefined micros, AREGSIZE and PREGSIZE:

```
A  =    "AREGSIZE"      ; Size of the A registers.
   CON  A               ; Store value in memory.
B  =    "PREGSIZE"      ; Size of the Program register.
   CON  B               ; Store value in memory.
```

3.12 Defined Sequences

The defined sequences pseudo instructions are as follows:

Pseudo	Description
MACRO	A sequence of source program instructions saved by the assembler for inclusion in a program when called for by the macro name. The macro call resembles a pseudo instruction.
OPDEF	A sequence of source program instructions saved by the assembler for inclusion in a program called for by the OPDEF pseudo instruction. The opdef resembles a symbolic machine instruction.
DUP	Introduces a sequence of code that is assembled repetitively a specified number of times; the duplicated code immediately follows the DUP pseudo instruction.
ECHO	Introduces a sequence of code that is assembled repetitively until an argument list is exhausted.
ENDM	Ends a macro or opdef definition.
ENDDUP	Terminates a DUP or ECHO sequence of code.
STOPDUP	Stops the duplication of a code sequence by overriding the repetition condition.
LOCAL	Specifies unique strings that are usually used as symbols within a MACRO, OPDEF, DUP, or ECHO pseudo instruction.

OPSYN	Defines a label field name that is the same as a specified operation in the operand field name.
EXITM	Terminates the innermost nested MACRO or OPDEF expansion.

3.13 File Control

The file control pseudo instruction, INCLUDE, inserts a file at the current source position. The INCLUDE pseudo instruction always prepares the file for reading by opening it and positioning the pointer at the beginning.

You can use this pseudo instruction to include the same file more than once within a particular file.

You can also nest INCLUDE instructions. Because you cannot use INCLUDE recursively, you should review nested INCLUDE instructions for recursive calls to a file that you have already opened.

CAL Program Organization [4]

The CAL program consists of sequences of source statements organized into program modules, program segments, and sections. This chapter describes the organization of a CAL program. Source statements are described in detail in Chapter 6, page 67.

The end of the chapter includes some small sample programs that provide the basic structure for getting started in writing CAL programs.

4.1 Program Modules

A *program module* is the main body of source statements and resides between the IDENT and END pseudo instructions. (For more information on pseudo instructions, see Chapter 3, page 33 and Chapter 8, page 129.) The IDENT pseudo instruction marks the beginning of a program module. The END pseudo instruction marks the end of a module. Any definitions between these two pseudo instructions apply only to the program module in which the definition resides.

4.2 Global Definitions and Local Definitions

CAL definitions are pseudo instruction source statements that do not generate code; they define and assign values to symbols, macros, opdefs, and micros. (For information on symbols, see Section 5.3.9.1, page 64. For information on macros, opdefs, and micros, see Section 5.3.2.2.3, page 56.)

CAL definitions that occur before the first IDENT pseudo instruction or between the END pseudo instruction that terminates one program module and the IDENT that begins the next program module, and are not the exceptions noted below, are *global definitions.* A global definition can be referenced without redefinition from within any of the program segments that follow the definition. Symbols defined within the global definitions area cannot be qualified.

CAL definitions that occur between an IDENT pseudo instruction and the following END pseudo instruction are *local definitions.* A local definition can be referenced only from within the program module in which it was defined.

The exceptions are redefinable micros, redefinable symbols, and symbols of the form $$x where x is 0 or more identifier characters. These definitions are local to the program segment in which they were defined, even if they were defined

within the global definition area. They are not known to the assembler after the END pseudo instruction that terminates that program segment, and they are not included in the cross-reference listing.

4.3 Program Segments

A program segment consists of global definitions, a program module, or a combination of global definitions and a program module. The END statement that terminates a module always terminates the segment containing it.

4.4 Program

The CAL program consists of one or more segments. Figure 1, page 43 illustrates the organization of the CAL program.

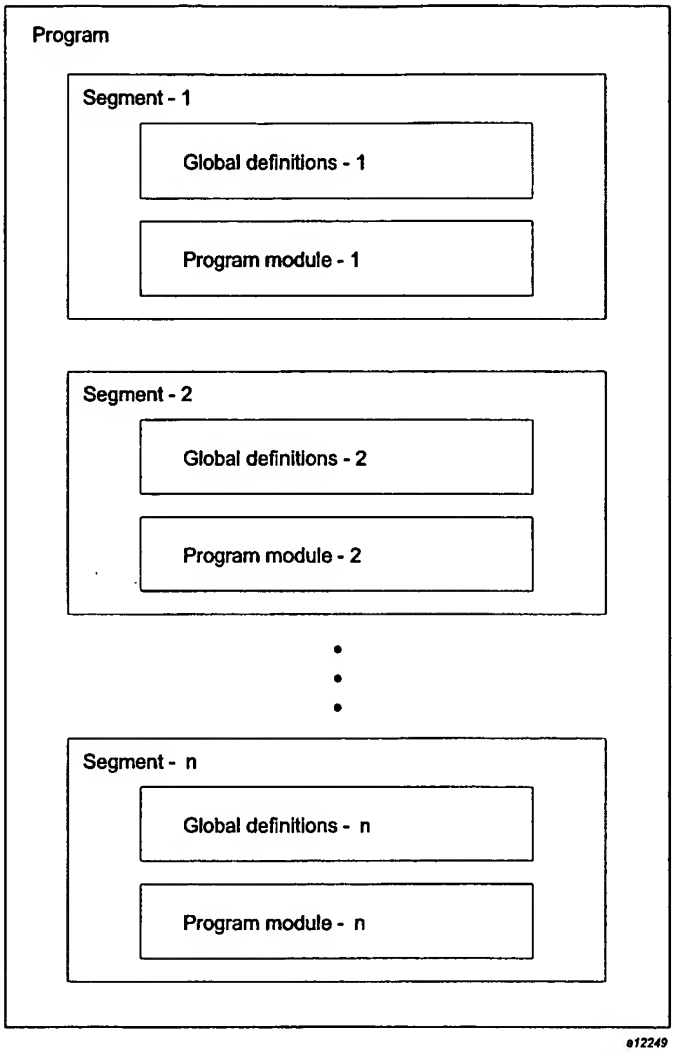

Figure 1. CAL Program Organization

4.5 Sections

A CAL program module can be divided into blocks of memory called *sections.* By dividing a module into sections, you can conveniently separate sequences of code from data. As the assembly of a program progresses, you can explicitly or implicitly assign code to specific sections or reserve areas of a section. The assembler assigns locations in a section consecutively as it encounters instructions or data destined for that particular memory section.

Use the main and literals sections for implicitly assigned code. The assembler maintains a stack of section names assigned by the SECTION pseudo instruction. All sections except stack sections are passed directly to the loader.

Sections can be local or common. A *local section* is available to the CAL program module in which it resides. A *common section* is available to another CAL program module.

To assign code explicitly to a section, use the SECTION pseudo instruction. For more details, see Section 8.53, page 191.

4.6 Examples

This section includes examples that show the basic structure of CAL programs.

4.6.1 Example 1: Global and Local Definitions

The following example illustrates a CAL program and the definition and the use of symbols in the context of segments and modules.

```
SYM1 = 1                ; Begin segment 1 global
                        ; SYM1 cannot be redefined
SYM2 SET 2              ; SYM2 equals 2 for this module
%%SYM3 = 3              ; Gone at the end of the module
%%SYM4 SET 4            ; Gone at the end of the module

       IDENT TEST1      ; Beginning of module 1
       S1  SYM1         ; Register S1 gets 1
       S2  SYM2         ; Register S2 gets 2
       S3  %%SYM3       ; Register S3 gets 3
       S4  %%SYM4       ; Register S4 gets 4
       END              ; End of segment 1 and module TEST 1

SYM2 SET   3            ; Beginning of segment 2
```

```
%%SYM3 =   5          ; Global definitions
       IDENT TEST2    ; Beginning of module TEST 2
       S1  SYM1       ; Register S1 gets 1
       S2  SYM2       ; Register S2 gets 3
       S3  %%SYM3     ; Register S3 gets 5
       S4  %%SYM4     ; Error:  not defined
       END            ; End of segment 2 and module TEST 2

       IDENT TEST3    ; Beginning of segment 3 and module TEST 3
       S1  SYM1       ; Register S1 gets 1
       S2  SYM2       ; Error:  not defined
       S3  %%SYM3     ; Error:  not defined
       END            ; End of segment 3 and module TEST 3
```

4.6.2 Example 2: Sections and Qualifiers

The following example illustrates section specification and deletion and indicates the current section. The example includes the QUAL pseudo instruction. For a description of the QUAL pseudo instruction, see Section 8.52, page 189.

```
IDENT   STACK          ; The IDENT statement puts the first entry
                       ; on the list of qualifiers. This entry
                       ; starts the symbol table for unqualified
                       ; symbols.
SYM1  =     1          ; SYM1 is relative to the main section.
      QUAL  QNAME1     ; Second entry on the list of qualifiers.
SYM2  =     2          ; SYM2 is the first entry in the symbol
                       ; table for QNAME1.
SNAME SECTION  MIXED   ; SNAME is the second entry on the list of
                       ; sections
      MLEVEL  ERROR    ; Reset message level to error eliminate
                       ; warning level messages.
SYM3  =     *          ; SYM3 is the second entry in the symbol
                       ; table for QNAME1 and is relative to the
                       ; SNAME section.
     MLEVEL  *         ; Reset message level to default in effect
                       ; before the MLEVEL specification.
     SECTION *         ; SNAME is deleted from the list of
                       ; sections.
SYM4  =     4          ; SYM4 is the third entry in the symbol
                       ; table for QNAME1 and is relative to the
                       ; main section.
      QUAL  QNAME2     ; Third entry on the list of qualifiers.
```

```
SYM5  =      5          ; SYM5 is the first entry in the symbol
                        ; table for QNAME2.
SYM6  =   /QNAME1/SYM2  ; SYM6 gets SYM2 from the symbol table for
                        ; QNAME1 even though QNAME1 is not the
                        ; current qualifier in effect.
      QUAL   *          ; QNAME2 is removed as the current
                        ; qualifier name.
SYM7  =      6          ; SYM7 is the fourth entry in the symbol
                        ; table for QNAME1.
SYM8  =      7          ; Second entry in the symbol table for
                        ; unqualified symbols.
```

CAL Assembler Invocation [5]

This chapter describes the as command and its options for invoking the CAL assembler.

5.1 Assembler Command Line

The as(1) command invokes the CAL assembler. The format of the as(1) command is:

```
as
   [-b bdflist]
   [-B]
   [-c bdfile]
   [-C cpu]
   [-D micdef]
   [-f]
   [-F]
   [-g symfile]
   [-G]
   [-h]
   [-H]
   [-i nlist]
   [-I options]
   [-j]
   [-J]
   [-l lstfile]
   [-L msgfile]
   [-m mlevel]
   [-M [no]message=n]
   [-n number]
   [-o objfile]
   [-U]
   [-v]
   [-V]
   [-x]
   file
```

The as(1) command assembles the specified file. The following options, each a separate argument, can appear in any order, but they must precede the *file* argument:

`-b` *bdflist*

Reads the binary definition files specified by *bdflist*. *bdflist* can be in one of the following forms:

- List of files separated by a comma
- List of files enclosed in double quotation marks and separated by a comma and/or one or more spaces

The assembler first reads the default binary definition file, unless suppressed by the `-B` option, and then reads the files in the order specified in bdflist. For details on binary definition files, see Section 5.3.2, page 54.

`-B`

Suppresses reading of the default binary definition file. The default is to read the default binary definition file.

`-c` *bdfile*

Creates binary definition file *bdfile*. By default, the assembler does not create a binary definition file. See Section 5.3.9, page 63.

`-C` *cpu*

Generates code for the specified CPU. By default, code is generated for the CPU specified by the TARGET environment variable. The `-C` option overrides the TARGET environment variable. If neither the TARGET environment variable nor the `-C` option are specified, the assembler generates code for a generic Cray X1 system.

cpu can be in one of the following forms:

primary [, *charc*]
charc

where *primary* is `cray-x1` and *charc* is a comma-separated list of CPU characteristics (currently, no CPU characteristics are supported).

`-D` *micdef*

Defines a globally defined constant micro *mname*, as follows:

micdef is *mname*[=[*string*]]

mname must be a valid identifier. If the = character is specified, it must immediately follow *mname*. The *string* that immediately follows the = character, if any, is associated with *mname*. If you do not specify *string*, then *mname* will be associated with an empty string.

If *mname* was defined as a micro by use of a binary definition file, the *mname* specified on the command line overrides the *mname* defined within the binary definition file; in that case, the assembler issues a note-level diagnostic message.

`-f`

Enables new statement format (default).

`-F`

Enables old statement format.

`-g` *symfile*

Creates assembly output symbol file *symfile*. By default, the assembler does not create an output symbol file.

`-G`

Forces all symbols to *symfile*. By default, unreferenced symbols are excluded.

`-h`

Enables all list pseudo instructions, regardless of location field name.

`-H`

Disables all list pseudos, regardless of location field name.

`-i` *nlist*

Restricts list pseudo processing to pseudos with location field names that are in *nlist*.

`-I` *options*

Lists options; overrides the list pseudo instruction. Specifying conflicting options is prohibited. Can be one or more options in any order without intervening spaces.

`b`	Enables `macro/opdef/dup/echo` expansion binary only
`B`	Disables `macro/opdef/dup/echo` expansion binary only (default)
`c`	Enables `macro/opdef/dup/echo` expansion conditionals
`C`	Disables `macro/opdef/dup/echo` expansion conditionals (default)
`d`	Enables `dup/echo` expansions
`D`	Disables `dup/echo` expansions (default)
`e`	Enables edited statement listing (default)
`E`	Disables edited statement listing
`h`	Enables page header in listings
`H`	Disables page header in listings (default)
`l`	Enables listing control pseudo instructions
`L`	Disables listing control pseudo instructions (default)
`m`	Enables `macro/opdef` expansions
`M`	Disables `macro/opdef` expansions (default)
`n`	Enables inclusion of nonreferenced local symbols in the cross-reference listing (default)
`N`	Disables inclusion of nonreferenced local symbols in the cross-reference listing
`p`	Enables `macro/opdef/dup/echo` expansion of pre-edited lines
`P`	Disables `macro/opdef/dup/echo` expansion of pre-edited lines (default)
`s`	Enables source statement listing (default)
`S`	Disables source statement listing
`t`	Enables text source statement listing
`T`	Disables text source statement listing (default)

x	Enables cross-reference listing (default)
X	Disables cross-reference listing

`-j`

Enables editing. Editing is enabled by default.

`-J`

Disables editing. Editing is enabled by default.

`-l` *lstfile*

Writes source statement listing to *lstfile*. By default, the assembler does not generate a source statement listing.

`-L` *msgfile*

Writes diagnostic message listing to *msgfile*. By default, the assembler does not generate a diagnostic message listing.

`-m` *mlevel*

Sets the message level for diagnostic listing, diagnostic message listing, and the standard error file. *mlevel* must be one of the following:

```
comment, note, caution, warning, or error
```

Listing messages of the specified *mlevel* and higher are issued. The default message level is `warning`.

`-M [no]message=`*n*`[:`*n*`]`

Enables or disables specified diagnostic messages. *n* is the number of the message to be enabled or disabled. You can specify more than one message number; multiple numbers must be separated by a colon with no intervening spaces. For example, to disable messages `as-17` and `as-74`, specify:

```
-M nomessage=17:74
```

This option overrides the `-m` *mlevel* option for the specified messages. If *n* is not a valid message number, it is ignored. Only messages of level `comment`, `note`, `caution`, and `warning` are affected by this option. Messages of level `error` or `internal` are not affected.

`-n` *number*

Sets the maximum *number* of diagnostic messages inserted into the listing files. Files used are the source statement listing, diagnostic message listing, and the standard error file. *number* must be 0 or greater; the default is 100.

`-o` *objfile*

The relocatable object file is *objfile*. By default, the relocatable object file name is formed by removing the directory name, if any, and the `.s` suffix, if any, from the input file and by appending the `.o` suffix.

`-U`

Forces source code to be converted to uppercase. Quoted strings are protected, as are embedded micros. Both new and old format statement types are supported.

`-V`

Writes the assembler version to standard error.

`-x`

Assemble instructions in interactive input/output mode. In this mode, the assembler takes as input from standard input a symbolic machine instruction and writes to standard output the 32-bit hex form. This can be used as input to the debugger to modify code being debugged.

Binary definition files are not processed. No instruction labels, macros, opdefs or micros are allowed. The interactive session is terminated by a `ctrl-D`.

file

Assembles the named file; all options must precede the *file* argument.

5.2 Environment Variables

The following subsections describe environment variables used by the assembler. How the environment variables are set depends on the type of shell being used.

5.2.1 ASDEF Environment Variable

The environment variable ASDEF specifies the full pathname of the default binary definition file. This environment variable is normally set by the `cal` modulefile in the programming environment but the user may redefine the variable. If ASDEF is not set, the assembler uses the pathname relative to the assembler executable (`../lib/asdef`), normally installed with the assembler, as the default path to binary definition file.

5.2.2 LPP Environment Variable

The LPP environment variable sets the number of lines per page for the source statement listing file and diagnostic message file. The valid range is 4-999. If the specified value is outside this range, the value is set to the default value. By default, the number of lines per page is 55.

5.2.3 TMPDIR Environment Variable

The TMPDIR shell variable specifies a directory that the assembler uses for its temporary file. If the directory is not specified or is specified incorrectly, the assembler uses the system default. The default is site specific.

5.2.4 TARGET Shell Variable

The TARGET environment variable determines the characteristics of the machine for which the code is generated.

The value of TARGET can be in one of the following forms:

primary [, *charc*]
, *charc*

where *primary* is `cray-xl` and *charc* is a comma separated list of CPU characteristics (currently, no cpu characteristics are supported).

5.3 Assembler Execution

When the assembler executes, it reads the specified source file and optionally reads one or more binary definition files. The source file may have one or more INCLUDE pseudo instructions which will cause the assembler to also read those source files.

The assembler makes two passes for each program segment. During the first pass, specified binary definition files are read, each source statement in the source file and included files is read, defined sequences (such as macros) are expanded, and memory addresses are assigned. During the second pass, values are substituted for the symbolic operands and addresses, any appropriate diagnostic messages are written, listing files, if specified, are written, and if there are no errors of ERROR or INTERNAL severity, an object file is generated. If specified, a binary definition file is also generated.

5.3.1 Reading Source Files

The assembler takes one source file as the primary input for assembly. The source file may contain more than one program segments. See Chapter 4, page 41.

5.3.2 Using Binary Definition Files

If one or more binary definition files is specified, the assembler reads the binary definition files before reading the source file. The global definitions (symbols, micros, macros, opdefs, and opsyns) in the binary definition file are then visible to the assembler as it processes the source statements in the source file.

> Note: System- and user-defined binary definition files are identical in all respects. Both types of files are created and used in exactly the same manner. In this manual, they are treated as separate entities to encourage you to define binary definition files that meet your particular programming requirements.

You can create user-defined binary definition files by using either of the following methods:

- Copying the system-defined binary definition files and then modifying the new file either by adding new definitions or by redefining existing definitions.
- Disabling the recognition of system-defined binary definition files and accumulating the defined sequences entirely from an assembler source program.

You can specify more than one binary definition file with each assembly. If more than one binary definition file is specified, the files are processed from left to right in the order specified by the `-b` option.

Lines or sequences of code assembled and stored in a binary definition file, can be accessed without reassembly. This means accessing a binary definition file directly saves assembler time.

When binary definition files are read, they are checked for the following:

- CPU compatibility
- Multiple references to the same definition

5.3.2.1 CPU Compatibility Checking

The assembler permits access to any previously defined file with one restriction. Binary definition files are marked with the CPU type for which they were created. Binary definition files created on one Cray system is not necessarily compatible with all Cray systems. If a binary definition file is not compatible with the system you are using, the binary definition file is not accepted and the following message is issued:

```
Incompatible version of binary definition file 'file'
```

This check ensures that the machine on which the binary definition file was created is compatible with the program trying to use it. Some CAL pseudo instructions have restricted use based on hardware and software requirements. The binary definition file compatibility check prevents the mixing of binary definition files and ensures that hardware and software restrictions are not violated.

5.3.2.2 Multiple References to a Definition

The assembler checks for multiple references to definition names for macros and opsyns, location field names for symbols and micros, and syntax for opdefs. The following subsections describe how multiple references to a definition are resolved.

5.3.2.2.1 Symbols

If a symbol is defined in more than one binary definition file, the definitions are compared. If the definitions are identical, the assembler disregards the duplicates and makes one entry for the symbol from the binary definition files. If a symbol is defined more than once and the definitions are not identical, the assembler uses the last definition associated with the location field name and issues the following diagnostic message:

```
Symbol 'name' is redefined in file 'file'
```

5.3.2.2.2 Macros

If a macro with the same name is defined in more than one binary definition file, the definitions are compared. If the definitions associated with the macro's name are identical character by character, the assembler disregards the duplicate definition and makes one entry for the macro from the binary definition files. If the name of the macro is used more than once, and the definitions associated with the name are not identical character by character, the assembler uses the definition associated with the last reference to the name and issues the following diagnostic message:

`Macro` *name* `in file`*file* `replaces previous definition`

If a macro is defined with the same name as a pseudo instruction, the macro replaces the pseudo instruction and the assembler issues the same message as shown above.

5.3.2.2.3 Opdefs

If an opdef with the same syntax is defined in more than one binary definition file, the definitions of the opdefs are compared. If the definitions of the two opdefs are exactly the same, the assembler disregards the duplicate definition and makes one entry for the opdef from the binary definition files. If the same syntax appears more than once and the definitions are not exactly the same, the syntax associated with the last reference to the opdef is used as its definition and the assembler issues the following diagnostic message:

`Opdef name in file` *file* `replaces previous definition`

If an opdef is defined with the same syntax as a machine instruction, the opdef replaces the machine instruction and the assembler issues the message shown above.

5.3.2.2.4 Opsyn

If an opsyn with the same name is defined in more than one binary definition file, the definitions are compared. If the definitions are identical, the assembler disregards the duplicate definition and makes one entry for the opsyn from the binary definition files. If the name for an opsyn is used more than once and the definitions are not identical, the assembler uses the definition associated with the last reference to the opsyn name and issues the following diagnostic message:

`Opsyn name in file` *file* `replaces previous definition`

If an opsyn is defined with the same name as a pseudo instruction, the opsyn replaces the pseudo instruction and the assembler issues the message as shown above. Pseudo instructions have an internal code that permits the assembler to identify them when they are encountered. When an opsyn is used to redefine an existing pseudo instruction, the assembler copies the predefined internal code of that pseudo instruction and uses it for identification in the binary definition file.

5.3.2.2.5 Micros

If a micro with the same location field name is defined in more than one binary definition file, the micro strings associated with the location field names are compared. If the strings are identical, the assembler disregards the duplicate definition and makes one entry for the micro from the binary definition files. If the micro is used more than once and the strings associated with the micro names are not exactly identical, the assembler uses the string associated with the last reference to the micro name and issues the following diagnostic message:

```
Micro 'name' in file 'file' replaces previous definition
```

5.3.3 Included Files

The `INCLUDE` pseudo instruction inserts a file at the current source position. The `INCLUDE` pseudo instruction always prepares the file for reading by opening it and positioning the pointer at the beginning.

5.3.4 Source Statement Listing File

The assembler generates a source statement listing and a cross-reference listing. You can control the format of these listings by using the listing control pseudo instructions or by using the `-i`, `-I`, `-l`, `-L`, `-a`, `-n`, `-h`, and `-H` options on the `as`(1) command line (see Section 5.1, page 47).

Each page of listing output produced by the assembler contains three header lines. Figure 1 shows the format of the page header.

Assembler version #	Title	Global page #
Date and time	Subtitle	Local page #
Section and qualifier	Scale (1-72 characters wide)	Cray Inc. System

a12245

Figure 2. Page Header Format

The three lines of the page header are described as follows:

- The first line contains, from left to right, the assembler version number, the title of the program, and a page number that is global over all programs assembled by the current assembly. If you do not specify a title by using a TITLE pseudo instruction, the title is taken from the operand field of the IDENT pseudo instruction.
- The second line contains, from left to right, the date and time of assembly, a subtitle if specified with a SUBTITLE pseudo instruction, and a page number that is local for this listing.
- On the third line, the left-most entry is a local section name if specified in a SECTION pseudo instruction. To the right of the local section name is a symbol qualifier name if specified by a QUAL pseudo instruction. The next field is a horizontal scale that is 72 characters wide, numbered from 1 through 72. This scale appears directly over your source code and helps you to differentiate the four fields of your source statements. On the far right of the third line is the name of the Cray Inc. system for which the code was generated.

5.3.4.1 Source Statement Listing

The format of the source statement listing, as shown in Figure 3, appears directly under the page header and contains five columns of information, as follows:

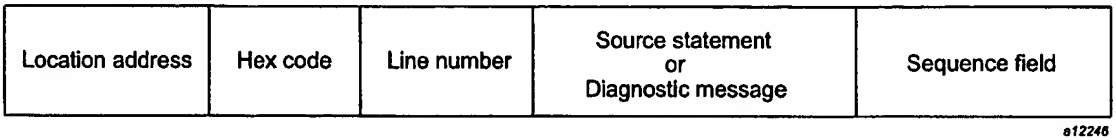

Figure 3. Source Statement Listing Format

The five columns of the source statement listing are described as follows:

- The *location address* column contains the hex representation of the byte address of the current statement.
- The *hex code* column contains the hex representation of the current instruction or numeric value.

  If the numeric value is an address, the hex code has one of the following suffixes:

  - + (Relocation in program block)
  - C (Common section)
  - D (Dynamic section)
  - S (Stack section)
  - T (Task common)
  - Z (Zero common)
  - : (Immobile attribute)
  - X (External symbol)
  - None (Absolute address)
- The results of several pseudo instructions can also appear in the hex code column:
  - The hex value of symbols defined by the SET, MICSIZE, or = pseudo instruction
  - The hex value of the number of bytes reserved by the BSS or BSSZ pseudo instruction

  - The hex value of the number of bytes skipped as a result of the ALIGN pseudo instruction
  - The hex value of the number of characters in a micro string defined by a MICRO, OCTMIC, or DECMIC pseudo instruction
- The *line number* column contains the line number of the source code.
- The *source statement* column is 72 characters wide and holds columns 1 through 72 of each source statement.
- The *diagnostic* column contains a diagnostic message immediately following a statement that contains an error.
- The *sequence field* column contains either an identifier or information taken from columns 73 through 90 of the source line image (which may be empty). It contains an identifier if the line is an expansion of a macro or opdef, or if the line was edited.

5.3.5 Cross-reference Listing

The assembler generates a cross-reference listing in the format shown in Figure 4. The assembler lists symbols alphabetically and groups them by qualifier if the QUAL pseudo instruction has declared qualifiers. If qualifiers were declared, each new qualifier appears on a fresh page. The qualifier name appears on the third line of the page header.

The cross-reference listing does not include unreferenced symbols that are defined between TEXT and ENDTEXT pseudo instructions and it does not include symbols of the form %%*xxxxxx*; x is zero or more identifier characters.

Note: The cross reference page header is nearly identical to the page header of the assembler listing; the difference is that the string *Symbol cross reference* is printed out in the middle field of the third line of the cross-reference listing.

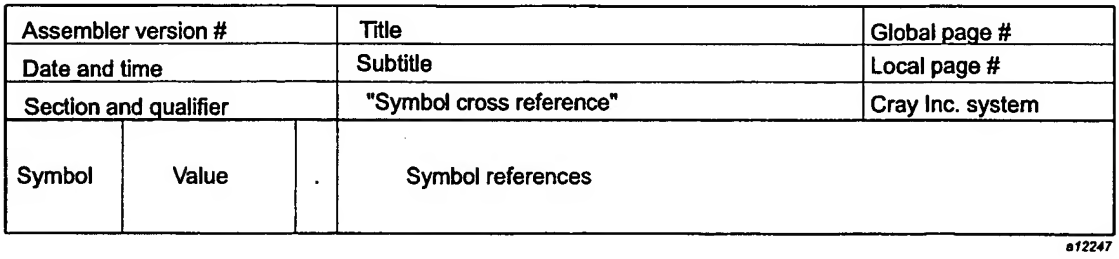

Figure 4. Cross-reference Listing Format

The information in each column is described as follows:

- The *symbol* column contains the symbol name.
- The *value* column contains the hex value of the symbol.

  The hex value of the symbol may have one of the following suffixes:

  - + (Relocation in program block)
  - C (Common section)
  - D (Dynamic section)
  - S (Stack section)
  - T (Task common)
  - Z (Zero common)
  - : (Immobile attribute)
  - X (External symbol)
  - None (Absolute address)
- The *period* (.) column separates the value field from the symbol reference fields and is called the separator.
- The *symbol references* column contains one or more references to the symbol.

  The assembler references symbols using the following format:

  page:line x

where *line* is the decimal representation of the line number that contains the reference.

x represents the type of reference as follows:

- A *blank* in this column means the symbol value is used at the specified line.
- D means the symbol is defined in the location field of an instruction or else by a SET, =, or EXT pseudo instruction.
- E means the symbol is an entry name.
- F means the symbol is used in an expression on an IFE, IFA, or ERRIF conditional pseudo instruction.
- R means the symbol is used in an address expression in a memory read instruction.
- S means the symbol is used in an address expression in a memory store instruction.

If a symbol is defined in text between TEXT and ENDTEXT pseudo instructions, the cross-reference listing reports the associated TEXT name on the line below the symbol reference.

If a symbol is defined in a binary definition file, the cross-reference listing reports the associated file name on the line below the symbol reference.

5.3.6 Diagnostic Messages

The assembler generates diagnostic messages that provide user information about the assembly. Diagnostic messages are classified by level of severity from low to high, as follows:

- Comment (statistical information)
- Note (possible assembly problems)
- Caution (definite user errors during assembly)
- Warning (possible error such as truncation of a value)
- Error (fatal assembly error, you should check the message and source code carefully for possible mistakes)

Note: To print comment-, note-, and caution-level diagnostic messages to the standard error file, you must specify the `-m` option on the `as`(1) command line.

- Internal (The assembler detected an error in the assembler itself. Assembly cannot continue.)

The diagnostic messages are written to the standard error file. If the `-l` option is specified, the diagnostic messages are also written to the source statement listing file immediately following the source statement that caused the diagnostic message.

5.3.7 Diagnostic Message Listing File

If the `-L` option is specified, the diagnostic messages are also written to the diagnostic message listing file.

5.3.8 Object File

The object file that is generated consists of a sequence of ELF tables. The loader processes the object file. For details on ELF files, see the `elf`(3e) man page.

5.3.9 Creating a Binary Definition File

To create binary definition files, include the `-b` and `-c` options on the `as`(1) command line. The `-b` option accepts a list of files separated by commas or a list of files enclosed in double quotation marks and separated by spaces or commas as arguments.

Only certain types of lines or sequences of code are permitted in a binary definition file. Binary definition files are always created from the global part of program segments and from any currently accessed binary definition files. Typically, binary definition files are created from source programs that include one segment that contains a global part, but has no program module (see Section 4.1, page 41).

Additions can be made to binary definition files from assembler source programs that include program modules, however, not all lines or sequences of code in the global part are added.

Note: Under no circumstance is any line or sequence of code added to a binary definition file from an assembler program module. All additions to binary definition files come from the global part of the segment.

Binary definition files are composed of lines or sequences of code classified as follows:

- Symbols
- Macros
- Opdefs
- Opsyns
- Micros

Each line or sequence of code added to a binary definition file must be in one of these classes and must satisfy the requirements for that particular class.

5.3.9.1 Symbols

The assembler accumulates the symbols to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of program segments that fit the following requirements:

- Symbols cannot be redefinable.

  To be included in a binary definition file, a symbol must be defined with the = (equates) pseudo instruction. Symbols defined with the `SET` or `MICSIZE` pseudo instruction are redefinable; therefore, they are not included in a binary definition file.

- Symbols cannot be preceded by `%%`.

  This exclusion applies to symbols that are created by the `LOCAL` and = pseudo instructions.

The assembler identifies all of the symbols in the global part of program segments that meet the preceding requirements and includes them in the creation of a binary definition file.

5.3.9.2 Macros

The assembler accumulates the macros to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

5.3.9.3 Opdefs

The assembler accumulates opdefs (operation definitions) to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

5.3.9.4 Opsyns

The assembler accumulates opsyns (operation synonyms) to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

5.3.9.5 Micros

The assembler accumulates micros to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within source program. Only micros that cannot be redefined are included in a binary definition file. A micro must be defined using the CMICRO pseudo instruction to be included in a binary definition file.

5.4 Linking

To create an executable program containing one or more assembler-generated object files, invoke the loader to link the object files with compiler- generated object files and/or object files contained in libraries. The loader may be invoked with commands `ld()`, `cc()`, or `ftn()`.

CAL Source Statements [6]

This chapter describes in detail the format and syntax for source statements, the components of source statements, and the attributes of symbols and expressions. The assembly language capabilities described here and in Chapter 9, page 213 provide very powerful programming capabilities beyond simply specifying symbolic machine instructions.

A CAL program is expressed in the form of source statements. CAL source statements fall into five categories:

- A symbolic machine instruction describes symbolically a Cray X1 machine instruction.
- A pseudo instruction controls the assembly process or results in generated data but does not result in generated machine instructions.
- A macro instruction represents a user-defined sequence of symbolic machine instructions and/or pseudo instructions.
- An opdef instruction represents a user-defined sequence of one or more symbolic machine instructions and has the same syntax as a symbolic machine instruction.
- A comment allows for annotation of a program but does not result in any generated machine instructions nor does it affect the assembly process.

The components of a statement are one or more of:

- A symbol is an identifier that has a value and several attributes.
- A tag is an identifier used in the label field of macros and other defined sequences but does not have a value or attributes.
- A mnemonic is an identifier that is a fixed part of a symbolic machine instruction or pseudo instruction.
- A constant is a character string that does not contain any operators and can be evaluated to a fixed value at assembly time.
- A data item specifies a numerical or character value in the operand field of the DATA pseudo instruction and in literals.
- A micro is an identifier that names a character string.
- A location element represents the current value of the location counter.

- An expression is a combination of one or more operands (constants, symbols, location elements) and zero or more operators that evaluates to a value with attributes. In CAL, registers are not a part of an expression, but registers and expressions with operators can be part of a symbolic machine instruction.

6.1 Source Statement Format

Each CAL source statement has three fields separated by white space and optionally followed by a comment.

Label *Result* *Operand* *Comment*

6.1.1 Label Field

The content of the label field depends on the requirements of the result and/or operand fields of each particular source statement. The label field of all machine instructions can optionally contain a symbol. If the label field of a machine instruction contains a symbol, the symbol value is set equal to the current value of the location counter. (See Chapter 4, page 41.)

When an instruction uses the label field, it begins in column 1 and is terminated by white space. The label field also can contain an asterisk (`*`) in column 1 to identify a comment line.

6.1.2 Result Field

The content of the result field depends on the particular instruction. The result field of pseudo instructions and macro instructions must match existing pseudo instructions or macros. Machine or opdef instructions can contain 1–5 subfields.

The subfield can be empty, contain expressions, or consist of registers or operators. The result field begins with the first non-whitespace character following a label field that is not empty and usually ends with white space or a semicolon. If column 1 is empty, the result field can begin in column 2 or subsequent columns. A blank result field following a label field produces a diagnostic message.

6.1.3 Operand Field

Before the operand field can be specified, it must be preceded by a result field.

If the statement is a symbolic machine instruction, the operand field contains the operation being performed. However, it can contain other information, depending on the particular instruction. The syntax of the operand field is identical to that of the result field. Machine or opdef instructions can contain one through five subfields.

Usually, the operand field begins with the first white space character following a result field that is not empty and ends with white space or a semicolon.

6.1.4 Comment Field

The comment field is optional and can be specified with an asterisk or a semicolon. A semicolon comment can be in any column, including column 1. If an asterisk is used to indicate a comment, it must appear in column 1. Generally, a comment that begins in column 1 is specified by using an asterisk and a comment that begins in any other column is specified by using a semicolon. Usually, comment fields are not edited. For more information about editing comment fields, see Section 6.10, page 97.

The following example illustrates the use of the comment field:

```
     ident  test1
*Asterisk in column 1 denotes comment line
                              ; Semicolon begins comment
     end    test1
```

6.1.5 Old Format

For compatibility with a prior generation of assemblers, an old source statement format is supported. The old format has the following differences:

- Only the space character is treated as a field separator.
- The semicolon character does not start a comment.
- The label field can start in either column 1 or column 2.
- A comma in column 1 indicates a continuation of the statement on the previous line.
- If the label field terminates before column 33, the result field must begin before column 35; otherwise, the field is considered empty.
- If the result field terminates before column 33, the operand field must begin before column 35; otherwise, the field is considered empty. If the result field

extends beyond column 32, however, the operand field must begin after not more than one blank separator and can begin after column 35.

- The comment field begins with the first nonblank character following the operand field or, if the operand field is empty, does not begin before column 35. If the result field extends beyond column 32 and no operand entry is provided, two or more blanks must precede the comment field.
- If editing is enabled, comments are edited.

The old format is specified by either the FORMAT pseudo instruction or the -F parameter of the assembler invocation statement.

6.1.6 Case Sensitivity

Formal parameters, symbols, names, and macro names are case sensitive. To be recognized, subsequent references to a previously defined formal parameter, symbol, name, or macro name must match the original definition character-for-character and case-for-case (uppercase or lowercase).

The following rules govern the use of uppercase and lowercase characters in CAL statements:

- Pseudo instructions and mnemonics are case sensitive; they can be uppercase or lowercase, but not mixed case.
- Register types are case insensitive; they can be uppercase, lowercase, or mixed case.
- Macro names, opdef mnemonics, symbol names, and other names are case sensitive; they are interpreted as coded.

6.2 Symbols

A *symbol* is an identifier that can be from 1–255 characters long and has an associated value and attributes. You can use symbols in expressions and in the following ways:

- In the label field of a source statement to define the symbol for use in the program and to assign it a value and certain characteristics called attributes.
- In the operand or result field of a source statement to reference the symbol.
- In loader linkage.

A symbol can be local or global depending on where the symbol is defined; that is, a symbol used within a single program module is local, and a symbol used by a number of program segments is global (see Section 4.2, page 41). A symbol also can be unique to a code sequence (see Section 6.2.1.2, page 72).

In addition to symbols specified in source statements, the assembler may generate symbols for internal use of the following form (where *n* is a decimal digit):

%%nnnnnn

Symbols that begin with the character sequence %% are discarded at the end of a program segment regardless of whether they are redefinable or defined in the global definitions part, and regardless of whether they are user-defined or generated by the assembler.

For more detailed information about symbols generated by the assembler, see the description of the LOCAL pseudo instruction in Section 8.38, page 177.

If a symbol name is the same as a Cray X1 register name, a warning message is issued.

6.2.1 Symbol Qualification

Symbols defined within a program module (between IDENT and END pseudo instructions) can be unqualified or qualified. They are unqualified unless preceded by the QUAL pseudo instruction (see Section 8.52, page 189, for more information).

6.2.1.1 Unqualified Symbol

The following statements describe ways in which unqualified symbols can be referenced:

- Unqualified symbols defined in an unqualified code sequence can be referenced without qualification from within that sequence.
- If the symbol has not been redefined within the current qualifier, unqualified symbols can be referenced without qualification from within the current qualifier.
- Unqualified symbols can be referenced from within the current qualifier by using the form *// symbol*.

Unqualified symbols are defined as follows:

symbol = *n* ; *symbol* is equal to *n*

The following example illustrates unqualified symbol definition:

```
       EDIT   OFF
       IDENT  TEST
SYM_1  =    *            ; SYM_1 has a value equal to the location
                         ; counter.
       A1   SYM_1        ; Register A1 gets SYM_1's value.
SYM_2  SET  2            ; SYM_2 is redefinable
SYM_3  =    3            ; SYM_3 is not redefinable.
       END
```

6.2.1.2 Qualified Symbols

You can make a symbol that is not a global symbol unique to a code sequence by specifying a symbol qualifier that will be appended to all symbols defined within the sequence.

Qualified symbols must be defined with respect to the following rules:

- A qualified symbol cannot be defined with a name that is reserved for registers.
- Symbols can be qualified only in a program module.

Qualified symbols can be referenced as follows:

- If a qualified symbol defined in a code sequence is referenced from within that sequence, it can be referenced without qualification.
- If a qualified symbol is referenced outside of the code sequence in which it was defined, it must be referenced in the form */ qualifier / symbol.* The *qualifier* variable is a 1- to 8-character identifier defined by the QUAL pseudo instruction and the *symbol* variable is a 1- to 255-character identifier.

Qualified symbols are defined as follows:

qualified_symbol = / [*identifier*] /*symbol*

The following example illustrates the use of qualified symbols:

```
      IDENT   TEST
SYM1  =     1              ; Assignment
      QUAL  NAME1          ; Declare qualifier name
```

```
SYM1  =     2              ; Qualified symbol SYM1
      S1    SYM1           ; Register S1 gets 2 (qualified SYM1)
      S1    //SYM1         ; Register S1 gets 1 (unqualified SYM1)
      S1    /NAME1/SYM1    ; Register S1 gets 2 (qualified SYM1)
      QUAL  *              ; Pop the top of the qualifier stack
      S1    SYM1           ; Register S1 gets 1
      S1    //SYM1         ; Register S1 gets 1
      S1    /NAME1/SYM1    ; Register S1 gets 2
      END
```

6.2.2 Symbol Definition

A symbol is defined by assigning it a value and attributes. The value and attributes of a symbol depend on how the program uses the symbol. The assignment can occur in the following three ways:

- When a symbol is used in the label field of a symbolic machine instruction or certain pseudo instructions, it is defined as follows:
  - It has the address of the current value of the location counter (for a description of counters, see Section 6.8.1, page 93).
  - It is an object or function entry point.
  - It has a byte-address attribute.
  - It is absolute, immobile, or relocatable.
  - It is not redefinable.
- A symbol used in the label field of a symbol-defining pseudo instruction is defined as having the value and attributes derived from an expression in the operand field of the instruction. Some symbol-defining pseudo instructions cause the symbol to have a redefinable attribute. When a symbol is redefinable, a redefinable pseudo instruction must be used to define the symbol the second time. Redefinition of the symbol causes it to be assigned a new value and attributes.
- A symbol can be defined as external to the current program module. A symbol is external if it is defined in a program module other than the module currently being assembled. The true value of an external symbol is not known within the current program module.

Here are examples of a symbol:

```
START  =    *              ; The symbol START has the current value of
```

```
                                ; the location counter and cannot be
                                ; redefined.
PARAM   SET   D'18              ; The symbol PARAM is equal to the decimal
                                ; value 18 and can be redefined.
        EXT   SECOND            ; Identifies SECOND as an external symbol.
```

6.2.3 Symbol Attributes

When a symbol is defined, it assumes two or more attributes. These attributes are in three categories:

- Type
- Relative
- Redefinable

Every symbol is assigned one attribute from each of the first two categories. Whether a symbol is assigned the redefinable attribute depends on how the symbol is used. Each symbol has a value of up to 64 bits associated with it.

6.2.3.1 Type Attribute

The following type attributes are supported:

- Object. This symbol represents the memory address at which data is stored.
- Function. This symbol represents the memory address of the beginning of a sequence of executable instructions.
- Value. This symbol represents a value that is not a memory address.

All memory addresses are byte addresses.

6.2.3.2 Relative Attributes

Each symbol is assigned one of the following relative attributes:

- Absolute

  A symbol is assigned the relative attribute of absolute when the current location counter is absolute and it appears in the label field of a machine instruction, BSS pseudo instruction, or data generation pseudo instruction such as BSSZ or CON or if it is equated to an expression that is absolute. All globally defined symbols have a relative attribute of absolute. The symbol is known only at assembly time.

- Immobile

  A symbol is assigned the relative attribute of immobile when the current location counter is immobile and it appears in the label field of a machine instruction, BSS pseudo instruction, or data generation pseudo instruction, such as BSSZ or CON, or if it is equated to an expression that is immobile. The symbol is known only at assembly time.

- Relocatable

  A symbol is assigned the relative attribute of relocatable when the current location counter is relocatable and it appears in the label field of a machine instruction, BSS pseudo instruction, or data generation pseudo instruction such as BSSZ or CON. A symbol also is relocatable if it is equated to an expression that is relocatable.

- External

  A symbol is assigned the relative attribute of external when it is defined by an EXT pseudo instruction. An external symbol defined in this manner is entered in the symbol table with a value of 0. The address attribute of an external symbol is specified as value (V), byte (B), or word (W); the default is value.

  A symbol is also assigned the relative attribute of external if it is equated to an expression that is external. Such a symbol assumes the value of the expression and can have an attribute of byte address, word address, or value.

  Note: The assignment of an unknown variable with a register at assembly time is made by using a symbol with a relative attribute of external.

In the following example, register S1 is loaded with variable EXT1 at assembly time:

```
      ident test1
      ext   test1      ; Variable ext1 is defined as an external
                       ; variable
      s1    ext1       ; ext1 transmits value to register s1
      end
      ident  test2
      entry  ext1
ext1  =      3         ; When the two modules are linked, register
                       ; S1 gets 3.
      end
```

6.2.3.3 Redefinable Attributes

In addition to its other attributes, a symbol is assigned the attribute of *redefinable* if it is defined by the SET or MICSIZE pseudo instructions. A redefinable symbol can be defined more than once in a program segment and can have different values and attributes at various times during an assembly. When such a symbol is referenced, its most recent definition is used by the assembler. All redefinable symbols are discarded at the end of a program segment without regard to whether they were defined in the global definitions.

The following example illustrates the redefinable attribute:

```
        IDENT   TEST
SYM1    =       1           ; Not redefinable
SYM2    SET     2           ; Redefinable
SYM1    SET     2           ; Error: SYM1 previously defined as 1
SYM2    SET     3           ; Redefinable
        END
```

6.2.4 Symbol Reference

When a symbol is in a field other than the label field, the symbol is being referenced. Reference to a symbol within an expression causes the value and attributes of the symbol to be used in place of the symbol. Symbols can be found in the operand fields of pseudo instructions.

An expression containing a symbol may have a different value and attributes depending on operators and other operands in the expression.

The following example illustrates a symbol reference:

```
S1      SYM1+1          ; Register S1 gets the value of SYM1+1.
                        ; SYM1+1 is an example of a symbol in an
                        ; operand field used in an expression.
IFA     DEF,SYM1        ; Symbols can also be used outside of an
                        ; expression. In this instance, SYM1 is
                        ; not used within an expression; it is a
                        ; symbol.
```

6.3 Tags

A tag is an identifier that does not have an associated value or attribute and cannot be used in expressions. Tags that are 1–32 characters in length are used to identify the following types of information:

- Macro instructions
- Micro character strings
- Conditional sequences
- Duplicated sequences

The first character must be one of the following:

- Alphabetic character (A through Z or a through z)
- Dollar sign ($)
- Percent sign (%)
- At sign (@)
- Underscore character (_)

Characters 2–32 can also be decimal digits (0 through 9).

Tags that are 1–255 characters in length can be used to identify the following types of information:

- Program modules
- Sections

The first character must be one of the valid tag characters or the underscore (_) character. Characters 2–255 can also be decimal digits (0 through 9).

Different types of tags do not conflict with each other or with symbols. For example, a micro can have the same tag as a macro, and a program module can have the same tag as a section.

Examples of valid and not valid tags:

Valid	Comment
`count`	Lowercase is permitted
`@ADD`	@ legal beginning character
`_SUBTRACT`	_ beginning character and 9 characters are legal
`ABCDE465`	Combinations of letters and digits are legal if the first character is legal

Not valid	Comment
`9knt`	Begins with a numeric character
`Y+Z3`	Contains an illegal character
`+YZ3`	Begins with +

Note: If you plan to use a source manager to store your CAL program, avoid using special character sequences such as the three-character string `%U%`. A source manager may replace these strings throughout your source program with other text. Because this type of string is allowed within identifiers and long-identifiers, avoid using it in names, long names, and symbols.

The underscore character (_) also is used as the concatenation character (see Section 6.10, page 97). Usually the assembler edits this character out of a source line. To insert this character into a long name, either disable editing or use the predefined concatenation micro (`$CNC`). To disable editing, use either the invocation statement or the `EDIT` pseudo instruction.

6.4 Constants

Constants can be defined as floating, integer, or character.

6.4.1 Floating Constant

A floating constant is evaluated as a 32-bit, 64-bit, or 128-bit quantity, depending on the precision specified.

The floating constant is defined as follows:

[*decimal-prefix*] *floating-decimal* [*floating-suffix*] [*binary-scale decimal-integer*]

In the preceding definition, variables are defined as follows:

- *decimal-prefix*

  This variable specifies the numeric base for the *floating-decimal* and/or the *decimal-integer* variables. `D'` or `d'` specifies a *decimal-prefix* and is the only prefix available for a floating constant.

- *floating-decimal*

  The *floating-decimal* variable can include the *decimal-integer*, *decimal-fraction* and/or *decimal-exponent* variables. A *decimal-integer* is a nonempty string of

decimal digits. A *decimal-integer* or a *decimal-fraction* is a nonempty string of decimal digits representing a whole number, a mixed number, or a fraction.

A *floating-decimal* can be defined as follows:

- A *decimal-integer* followed by a *decimal-fraction* with an optional *decimal-exponent* and *decimal-integer*. For example:

  n.*n* or *n*. *n*E*n* or *n*.*n*E+ *n* or *n*. *n*D*n* or *n*. D+*n*

- A *decimal-integer* followed by a period (.) with a *decimal-exponent* and *decimal-integer*. For example:

  n. or *n*.E or *n*. +*n* or *n*.*nn* or *nn*D+ *n*

- A *decimal-integer* followed by a *decimal-exponent* and *decimal-integer*. For example:

  *n*E*n* or *n* E or *n*D*n* or *n*D*n*

- A *decimal-fraction* followed by an optional *decimal-exponent* and *decimal-integer*. For example:

  .*n* or .*n*E*n* or .*n*E + *n* or .*n*D*n* or .*n*D*n*

- *decimal-exponent*

  The power of 10 by which the integer and/or *fraction* will be multiplied; indicates whether the constant will be single precision (E or e; one 64-bit word) or double precision (D or d; two 64-bit words). *n* is an integer in the base specified by *prefix*.

  If no *decimal-exponent* is provided, the constant occupies one word. *decimal-exponents* are defined as follows:

  - E *n* (Positive decimal exponent, 64-bit double precision)
  - E+ *n* (Positive decimal exponent, 64-bit double precision)
  - E- *n* (Negative decimal exponent, 64-bit double precision)
  - D *n* (Positive decimal exponent, 128-bit long double precision)
  - D+ *n* (Positive decimal exponent, 128-bit long double precision)
  - D- *n* (Negative decimal exponent, 128-bit long double precision)

- *floating-suffix* (one of *f* or FL)
- *binary-scale decimal-integer*

The *integer* and/or *fraction* will be multiplied by a power of 2. Binary scale is specified with S or s and an optional add-operator (+ or -). *n* is an integer in the base specified by the *decimal-prefix*. For example:

S*n* or S+*n*	Positive binary exponent
s*n* or s+*n*	Positive binary exponent
S-*n* or s-*n*	Negative binary exponent

Note: Double-precision floating-point numbers are truncated to single-precision floating-point numbers if pseudo instructions, which can reserve only one memory word (such as the CON pseudo instruction) are used.

The following examples illustrate floating constants:

```
        CON     D'1.5           ; Mixed decimal of the form n.n.
        CON     4.5E+10         ; Single-precision floating constant of
                                ; the form n.nE+n.
        CON     4.D+15          ; Double-precision floating constant of
                                ; the form n.D+n.
        CON     D'1.0E-6        ; Negative floating constant of the form
                                ; n.nE-n.
        CON     1000e2          ; Single precision floating constant of
                                ; the form nD+n.
SYM     =       1777752d+10     ; Double-precision floating constant of
                                ; the form nD+n.
```

6.4.2 Integer Constant

An *integer constant* is evaluated as a 64-bit twos complement integer. The integer constant is defined as follows:

base-integer [*binary-scalebase-integer*]

octal-prefix octal-integer [*binary-scale octal-integer*]

decimal-prefix decimal-integer [*binary-scale decimal-integer*]

hex-prefix hex-integer [*binary-scale hex-integer*]

In the preceding definition, variables are defined as follows:

- *base-integer*

A string of decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) of any length

- *binary-scale*

  The integer and/or fraction that will be multiplied by a power of 2. *binary-scale* is specified with S or s and an optional add-operator (+ or -). *n* is an integer in the base specified by the *decimal-prefix*. For example:

S*n* or S+*n*	(positive binary exponent)
s*n* or s+*n*	(positive binary exponent)
s-*n* or S-*n*	(negative binary exponent)

- *base-integer, octal-prefix, decimal-prefix,* or *hex-prefix*

  Numeric base used for the integer. If no prefix is used, base-integer is determined by the default mode of the assembler or by the BASE pseudo instruction. A prefix can be one of the following character combinations:

D' or d'	Decimal (default mode)
O' or o'	Octal
X', x', 0X, or 0x	Hexadecimal

- *octal-integer*

  A string of octal integers (0, 1, 2, 3, 4, 5, 6, 7) of any length

- *decimal-integer*

  A string of decimal integers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) of any length

- *hex-integer*

  A string of hexadecimal integers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A or a, B or b, C or c, D or d, E or e, F or f) of any length

The following examples illustrate integer constants:

```
        S1    O'1234567     ; Octal-prefix followed by octal-integer.
        A4    D'50          ; Integer-constant of the form
                            ; decimal-prefix followed by
                            ; decimal-integer.
SYM     =     x'ffffffa     ; Integer-constant of the form hex-prefix
                            ; followed by hex-integer.
SYM2    =     0Xbeef        ;
```

6.4.3 Character Constants

The character constant is defined as follows:

character-string [*character-suffix*]

In the preceding definition, variables are defined as follows:

- *character-string*

  The default is a string of zero or more characters (enclosed in apostrophes) from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate one apostrophe.

- *character-suffix*

  The justification and fill of a character string:

H or h	Left-justified, blank-filled (default)
L or l	Left-justified, zero-filled
R or r	Right-justified, zero-filled
Z or z	Left-justified, zero-filled, at least one trailing binary zero character guaranteed

The following examples illustrate character constants:

```
S3   '*'R        ; ASCII character, right justified, zero filled.
CON  'ABC'L      ; ASCII characters, left justified, zero filled.
S1   'XYZ'H      ; ASCII characters, left justified, blank filled.
CON  'OUT'       ; ASCII characters, left justified, blank
                 ; filled (default).
VWD  32/'EFG'    ; ASCII characters, left justified, blank
                 ; filled within a 32-bit field (all default).
```

6.5 Data Items

A *character* or *data* item can be used in the operand field of the DATA pseudo instruction and in literals. The length of the data field occupied by a data item is determined by its type and size. Data items can be floating, integer, or character. The sections that follow describe these types of data items.

6.5.1 Floating Data Item

Single-precision floating data items occupy one word and double-precision floating data items occupy two words. A floating data item is defined as follows:

```
[sign] floating-constant
```

In the preceding definition, the *sign* variable is defined as follows:

- *sign*

  The sign variable determines how the floating data item will be stored. The sign variable can be specified as follows:

+ or omitted	Not complemented
-	Negated (twos complemented)
~	Ones complemented

  Note: Although syntactically correct, ~ is not permitted; a semantic error is generated with floating data.

- *floating-constant*

  The syntax for a floating data item is the same as the syntax for floating constants. Floating constants are described in Section 6.4.1, page 78.

The following example illustrates floating constants for data items:

```
DATA  D'1345.567        ; Decimal floating data item
                        ; of the form n.n.
DATA  1345.E+1          ; Decimal floating data item
                        ; of the form n.E+n.
DATA  4.5E+10           ; Single-precision floating
                        ; constant of the form n.nE+n.
DATA  4.D+15            ; Double-precision floating
                        ; constant of the form n.D+n.
DATA  D'1.0E-6          ; Negative floating constant
                        ; of the form n.nE-n.
DATA  1000e2            ; Single-precision floating
                        ; constant of the form nen.
DATA  1.5S2             ; Floating binary scale data
                        ; item of the form n.nSn.
```

6.5.2 Integer Data Item

An integer data item occupies one 64-bit word and is defined as follows:

```
[ sign ] integer-constant
```

In the preceding definition the *sign* variable defines the form of a data item to be stored. The *sign* variable can be replaced in the integer data item definition with any of the following:

+ or omitted	Not complemented
-	Negated (twos complemented)
~	Ones complemented

The syntax for *integer-constant* is described in Section 6.4.2, page 80.

The following example illustrates integer constants for data:

```
DATA  +o'20          ; Octal integer
VWD   40/0,24/O'200
```

6.5.3 Character Data Item

The character data item is as follows:

[*character-prefix*] *character-string* [*character-count*] [*character suffix*]

In the preceding definition, variables are defined as follows:

- *character-prefix*

  This variable specifies the character set used for the stored constant. It is specified as follows:

  `A` or `a` ASCII character set (default)

- *character-string*

  The default is a string of zero or more characters (enclosed in apostrophes) from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate one apostrophe.

- *character-count*

  The length of the field, in number of characters, into which the data item will be placed. If *count* is not supplied, the length is the number of words

needed to hold the character string. If a count field is present, the length is the character count times the character width; therefore, length is not necessarily an integral number of words. The character width is 8 bits for ASCII or EBCDIC, and 6 bits for control data display code.

If an asterisk is in the count field, the actual number of characters in the string is used as the count. Two apostrophes that are used to represent one apostrophe are counted as one character.

If the base is mixed, the assembler assumes that the count is decimal

- *character-suffix*

  This variable specifies justification and fill of the character string as follows:

`H` or `h`	Left-justified, blank-filled (default)
`L` or `l`	Left-justified, zero-filled
`R` or `r`	Right-justified, zero-filled
`Z` or `z`	Left-justified, zero-filled, at least one trailing zero character guaranteed

The following example illustrates character data items:

```
DATA  A'ERROR IN DSN'  ; ASCII character set left justified and
                       ; blank fill by default; two words
DATA  A'error in dsn'R ; ASCII character set right justified,
                       ; zero filled; stored in two words.
DATA  'Error'          ; Default ASCII character set left
                       ; justified and blank filled by default
                       ; stored in one word.
```

6.6 Literals

Literals are read-only data items whose storage is controlled by the assembler. Specifying a literal lets you implicitly insert a constant value into memory. The actual storage of the literal value is the responsibility of the assembler. Literals can be used only in expressions because the address of a literal, rather than its value, is used.

The first use of a literal value in an expression causes the assembler to store the data item in one or more words in a special local block of memory known as the *literals section.* Subsequent references to a literal value do not produce multiple copies of the same literal.

Because literals can map into the same location in the literals section, the assembler checks for the presence of matching literals before new entries are added. This check is made bit by bit. If the current string is identical to any string currently stored in the literals section, the assembler maps that string to the location of the matching string. If the current string is not identical to any of the strings currently stored, the current string is considered to be unique, and is assigned a location in the literals section.

The following special syntaxes are in effect for literals:

- Literals always have the following attributes:
  - Relocatable (relative) to a constant section
  - Byte (address)
- Literals cannot be specified as character strings of zero bits. The actual constant within a literal must have a bit length greater than 0. In actual use, you must specify at least one 8-bit character for the ASCII character set.
- By default, literals always fall on full-word boundaries. Trailing blanks are added to fill the word to the next word boundary.

When used as an element of an expression, a literal is defined as follows:

=data-item

A data item for literals is the same as data items for constants. Data items for constants are described in Section 6.5, page 82.

Single-precision literals are stored in one 64-bit word (default). Double-precision literals are stored in two 64-bit words. The following example shows how literals can be specified with single or double precision:

```
CON    =1.5            ; Single-precision literal
CON    =1.sD1          ; Double-precision literal
```

Figure 5 illustrates how the ASCII character a is stored by either of the following instructions (^ represents a blank character):

```
CON ='a'H

CON ='a'
```

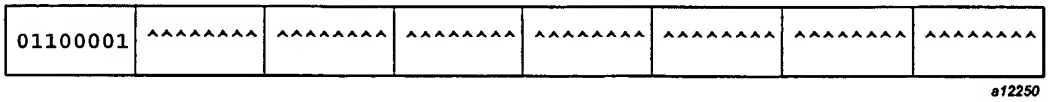

Figure 5. ASCII Character with Left-justification and Blank-fill

Figure 6 illustrates how the ASCII character a is stored by any of the following instructions:

```
CON ='a'L

CON 'a'R

CON -'a'S
```

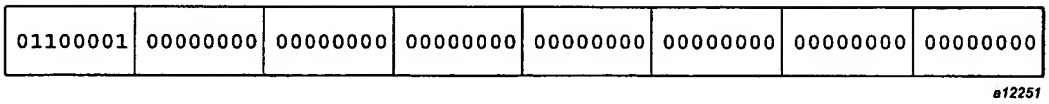

Figure 6. ASCII Character with Left-justification and Zero-fill

Figure 7 illustrates how the ASCII character a is stored by the following instruction:

```
CON ='a'R
```

This example illustrates how the ASCII character a is stored when `='a'R` is specified.

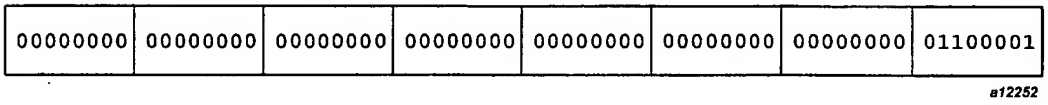

Figure 7. ASCII Character with Right-justification and Zero-fill

Figure 8 illustrates how the ASCII character a is stored by the following instruction:

```
CON ='a'*R
```

01100001	00000000	00000000	00000000	00000000	00000000	00000000	00000000

a12253

Figure 8. ASCII Character with Right-justification in 8 bits

The character set available to CAL is declared as follows:

```
CON ='A'    ; 8-bit ASCII character.
```

The following example illustrates the use of the H, L, R, or Z options when specifying literals:

```
CON  ='AB'3        ; Left-justified with one blank-padded on the
                   ; right (default).
CON  ='AB'3H       ; Left-justified with one blank-padded on the
                   ; right (default).
CON  ='AB'6R       ; Right-justified, filled with four leading
                   ; zeros.
CON  ='AB'6Z       ; Left-justified, padded with four trailing
                   ; zeros
```

6.7 Micros

Through the use of micros, you can assign a name to a character string and subsequently refer to the character string by its name. A reference to a micro results in the character string being substituted for the name before assembly of the source statement containing the reference. The CMICRO, MICRO, OCTMIC, and DECMIC pseudo instructions (described in Chapter 8, page 129) assign the name to the character string.

Refer to a micro by enclosing the micro name in double quotation marks (" ") anywhere in a source statement other than within a comment. If column 72 of a line is exceeded because of a micro substitution, the assembler creates additional continuation lines. No replacement occurs if the micro name is unknown or if one of the quotation marks is omitted.

When a micro is edited, the source statement that contains the micro is changed. Each substitution produces one of the following cases:

- The length of the micro name and the pair of quotation marks is the same as the predefined substitute string. When the micro is edited, the length of the source statement is unchanged.

- The length of the micro name and the double quotation marks is greater than the predefined substitute string. When the string is edited, all characters to the right of the edited string shift left the number of spaces equal to the difference between the length of the micro name including the double quotation marks and the predefined substitute string.
- The length of the micro name and the double quotation marks is less than the predefined substitute string. If column 72 of a line is exceeded because of a micro substitution, the assembler creates additional continuation lines. Resulting lines are processed as if they were one statement.

In the following example, the length of the micro name (including quotation marks) is equal to the length of the predefined substitute string. A micro named PFX is defined as EQUAL. A reference to PFX is in the label field of the statement, as follows:

```
"PFX"TAG  S0       S1    ; The location of S0 and S1 on the
                         ; source statement is unchanged
```

When the line is interpreted, the assembler substitutes EQUAL for "PFX", producing the following line:

```
EQUALTAG  S0       S1    ; The location of S0 and S1 on the
                         ; source statement is unchanged
```

In the following example, the length of the micro name (including quotation marks) is greater than the length of the predefined substitute string. A micro named PFX is defined as LESS. A reference to PFX is in the label field of the statement, as follows:

```
"PFX"TAG  S0       S1    ; Because LESS is one character shorter
                         ; than the micro string name "PFX", the
                         ; values in the result and operand
                         ; fields are shifted one space to the
                         ; left.
```

Before the line is interpreted, the assembler substitutes LESS for "PFX", producing the following line:

```
LESSTAG  S0        S1    ; Because LESS is one character shorter
                         ; than the micro string name "PFX", the
                         ; values in the result and operand
                         ; fields are shifted one space to the
                         ; left.
```

In the following example, the length of the micro name (including quotation marks) is less than the length of the predefined substitute string. A micro named `pfx` is defined as `greater`. A reference to `pfx` is in the label field of the following statement:

```
"pfx" tag S0        S1   ; Because greater is two characters
                         ; longer than micro string name "pfx",
                         ; the values in the result and operand
                         ; fields are shifted two spaces to the
                         ; right.
```

Before the line is interpreted, the assembler substitutes the predefined string `greater` for `"pfx"`. Because the predefined substitute string is 2 characters longer than micro name, the fields to the right of the substitution are shifted 2 characters to the right, producing the following statement:

```
greatertag  S0      S1   ; Because greater is two characters
                         ; longer than the micro string name
                         ; "pfx", the values in the result and
                         ; operand fields are shifted
```

One or more micro substitutions can occur between the beginning and ending quotation marks of a micro. These substitutions create a micro name that is substituted, along with the surrounding quotation marks, for the corresponding micro string. Substitutions of this type are *embedded micros*. An embedded micro consists of a micro name included between a left ({) and a right brace (}) and is specified as follows:

{*microname*}

When a micro that contains one or more embedded micros is encountered, the assembler edits all embedded micros within the micro until a micro name is recognized or until the micro name is determined to be illegal (undefined or exceeding the maximum allowable string length of 8 characters). When an illegal micro is encountered, the assembler issues an appropriate message and terminates the editing of the micro. An embedded micro also can contain one or more embedded micros.

The following example includes valid and not valid defined embedded micros

```
index   micro       \     ; Assigns literal value to index
null    micro        \\   ; Assigns literal value to null

array "index"  micro \Some string\
array1 micro \Some string\
```

```
* "array1" - an explicit reference
* Some string - an explicit reference; edited by the assembler
* "array" "index" - not valid, because "array" was not defined
* "array"1 - not valid, because "array" was not defined; edited by the assembler
* "array{index}" - This is an example of an embedded micro
* Some string - This is an example of an embedded micro; edited by the assembler
* "{null}array{index}" - This is an example of two embedded micros
* Some string - This is an example of two embedded micros; edited by the assembler
```

The assembler places no restrictions on the number of recursions that are necessary to identify a micro name. The following example demonstrates the unlimited recursive editing capability of the assembler on embedded micros:

```
index    micro      \1\  ; Assigns literal value to index
null     micro       \\  ; Assigns literal value to null
array"index" micro \Some string\
array1 micro \Some string\; edited by the assembler
* "{nu{n{null}u{null}ll}ll}ar{null{null}}ray{ind{null}ex}" - Micro
* Some string - Micro; edited by the assembler
```

The assembler issues a warning- or error-level listing message when an invalid micro name is specified. If a micro name is recognized as invalid before editing begins, a warning-level message is issued. If an embedded micro has been edited and the resulting string is not a valid micro name, an error-level listing message is issued.

The following examples demonstrate how the assembler assigns levels to messages when a micro that is not valid is encountered:

```
identity   micro    \The substitute string for this example\;
null    micro       \   ; Assigns literal value to null
* "identity{null}" - This is a valid micro
* The substitute string for this example - This is a valid
* micro; edited by the assembler
* The following micro is invalid, because the maximum micro
* name length of eight characters is exceeded.  When a micro
* name is identified as being invalid before editing occurs,
* a warning-level listing message is issued:
*   "identity9{null}" - This is a not valid micro; edited by the assembler
*   "identity9 - This is a not valid micro
* The following micro is not valid, because the maximum
* micro name length of eight characters is exceeded.  When a
* micro name is identified as being not valid after editing
```

```
* occurs, an error-level listing message is issued:
*   "id{null}entity9{null}" - This is a not valid micro
*   "identity9" - This is a not valid micro; edited by the assembler
```

6.8 Location Elements

Location elements are used to obtain the current value of the location counter, the origin counter, and the Longword-bit-position counter. Location elements can occur as elements of expressions. For a description of expression elements, see Section 6.9, page 94. The origin, location, word-bit-position, and byte-bit-position counters are described in Chapter 8, page 129.

The following elements have special meanings to the assembler:

- `*`. Location counter.

  The asterisk (`*`) denotes a value equal to the current value of the location counter with byte-address attribute and absolute, immobile, or relocatable attributes. The location counter is absolute if the LOC pseudo instruction modified it by using an expression that has a relative attribute of absolute. The location counter is immobile if it is relative to either a STACK or TASKCOM section. The location counter is relocatable in all other cases.

- `*A` or `*a`. Absolute location counter.

  The `*A` or `*a` denotes a value equal to the current value of the location counter with byte-address and absolute attributes.

- `*B` or `*b`. Absolute origin counter.

  The `*B` or `*b` denotes a value equal to the current value of the origin counter relative to the beginning of the section with byte-address and absolute attributes.

- `*O` or `*o`. Origin counter.

  The `*O` or `*o` denotes a value equal to the current value of the origin counter relative to the beginning of the current section. The origin counter has an address attribute of byte. If the current section is a section with a type of STACK or TASKCOM, it has an immobile attribute. In all other cases, it has a relative attribute of relocatable.

- `*W` or `*w`. Longword pointer.

  The `*W` or `*w` denotes a value equal to the current value of the Longword-bit-position counter with absolute and value attributes. `*W` is

relative to the word and the word-bit-position counter is almost always equal to 0, 16, 32, or 48. The assembler issues a warning message when the word-bit-position counter has a value other than 0 (not pointing at a word boundary) and is used in an expression.

6.8.1 Location Counter

Usually, the *location counter* is the same value as the origin counter and the assembler uses it to define symbolic addresses within a section. The counter is incremented when the origin counter is incremented. Use the LOC pseudo instruction to adjust the location counter so that it differs in value from the origin counter or so that it refers to the address relative to a section other than the one currently in use. When the location element * is used in an expression, the assembler replaces it with the current byte-address value of the location counter for the section in use. To obtain the 64-bit word-address value of the location counter, use W.*.

6.8.2 Origin Counter

The *origin counter* controls the relative location of the next word that will be assembled or reserved in the section. You can reserve blank memory areas by using either the ORG or BSS pseudo instructions to advance the origin counter.

When the location element *O is used in an expression, the assembler replaces it with the current byte-address value of the origin counter for the section in use. To obtain the word-address value of the origin counter, use W.*O. (In this context, a word is a 64-bit long word.).

6.8.3 Longword-bit-position Counter

As instructions and data are assembled and placed into a 64-bit long word, the assembler maintains a pointer that indicates the next available bit within the word currently being assembled. This pointer is known as the *word-bit-position counter.* It is 0 at the beginning of a new word and is incremented by 1 for each completed bit in the word. Its maximum value is 63 for the rightmost bit in the word. When a word is completed, the origin and location counters are incremented by 1, and the word-bit-position counter is reset to 0 for the next word.

When the location element *W is used in an expression, the assembler replaces it with the current value of the word-bit-position counter. The normal advancement of the word-bit-position counter is in increments of 32 as instructions are

generated. You can alter this normal advancement by using the BITW, BITP, DATA, and VWD pseudo instructions.

6.8.4 Force Longword Boundary

If either of the following conditions are true, the assembler completes a partial word and sets the word-bit-position counter to 0:

- The current instruction is an ALIGN, BSS, BSSZ, CON, LOC, or ORG pseudo instruction.
- The current instruction is a DATA or VWD pseudo instruction and the instruction has an entry in the label field.

6.9 Expressions

The result and operand fields for many source statements contain expressions. An expression consists of one or more constants (see Section 6.4, page 78), literals (see Section 6.6, page 85), location elements (see Section 6.8, page 92) or symbols (see Section 6.2, page 70) and zero or more operators (see Section 6.9.1, page 95).

There are some restrictions on the combinations of operands and operators that are allowed in the different contexts, as specified below.

The assembler will evaluate the expression, applying appropriate restrictions and if no restrictions are violated, insert the value of the expression into the immediate field in the machine instruction or the operand field of the pseudo instruction.

Registers are not included as part of a CAL expression. A register may be an operand in an instruction but the assembler cannot determine the value of the register. For example, in the instruction:

```
s5      s3+2*3
```

the assembler parses the instruction such that `2*3` is an expression that it evaluates and then inserts the value of `6` into the immediate field of the machine instruction. It does not treat `s3` as part of the expression.

Each CAL expression has a type and a value. The type is one of:

- absolute
- object

- function

If the type of the expression is either object or function, it has a *relative attribute* that is one of:

- relocatable
- external
- immobile

and has an *address attribute* that is one of:

- byte address
- stack offset

6.9.1 Operators

The CAL operators are:

- unary operators:
  - + (unary plus)
  - - (unary minus)
  - ~ (logical complement)
  - ! (logical NOT)
  - B. (byte prefix)
  - b. (byte prefix)
  - W. (word prefix)
  - w. (word prefix)
  - L. (longword prefix)
  - l. (longword prefix)
  - < (mask right)
  - > (mask left)
- binary operators:
  - + (add)

- - (subtract)
- * (multiply)
- / (divide)
- & (logical AND)
- | (logical OR)
- ^ (logical XOR)
- && (logical AND)
- || (logical OR)
- << (logical left shift)
- >> (logical right shift)

6.9.2 Operator Precedence

CAL expressions are evaluated according to the following operator precedence:

```
! ~ +(unary) -(unary) B. W. L. b. w. l. < >  Right to left
* /                                          Left to right
+ -                                          Left to right
<< >>                                        Left to right
&                                            Left to right
^                                            Left to right
|                                            Left to right
&&                                           Left to right
||                                           Left to right
```

Parentheses can be used to group an expression or subexpressions.

6.9.3 Restrictions

If the machine instruction has an immediate field (`imm6`, `imm8`, `imm14`, `imm16`, or `imm20`), then an expression is allowed in that subfield of the CAL instruction. If the value of the expression exceeds the size of the immediate field, the value will be truncated and a diagnostic message will be issued.

If a pseudo instruction has an expression field or subfield, the restrictions are described in Chapter 8, page 129. Only integer expressions are allowed in pseudo instructions unless explicitly stated otherwise.

If the expression includes one or more relocatable or external symbols that is known only at link time, the assembler will evaluate the expression including the symbol name and pass to the loader the symbol name and an offset value to allow the loader to complete the evaluation and insert the final value in the code of the executable.

A symbolic register designator may be an expression but may not include any relocatable symbols and the entire expression must be enclosed in parentheses.

Floating constants are allowed only in the CON pseudo instruction.

Two relocatable symbols can occur in the same expression only if the difference of the two symbols is taken.

6.10 Statement Editing

The assembler processes source statements sequentially from the source file. Statement editing is a form of preprocessing in which the assembler deletes or replaces characters before processing the statement as source code.

The assembler performs the following types of statement editing:

- Concatenation

  The assembler recursively deletes all underscore characters and combines the character that preceded the underscore with the character following the underscore.

- Micro substitution

  The assembler replaces a micro name with a predefined character string. The character string replacement is not edited a second time.

A macro or opdef definition is not immediately interpreted but is saved and interpreted each time it is called. Before interpreting a statement, the assembler performs editing operations. The assembler does not perform micro substitution or concatenate lines when editing is disabled. (Editing is disabled by using the EDIT pseudo instruction or by including the -J option in the invocation line of the assembler.)

The edit invocation statement option does not affect appending, continuation, and the processing of comments.

The following special characters signal micro substitution, concatenation, append, continuation, and comments:

Character	Edit	Description
" *name* "	Yes	Micro; affected by the EDIT pseudo instruction on the invocation statement option.
_	Yes	Concatenate; (underscore) affected by the EDIT pseudo instruction on the invocation statement option.
*	No	Comment line; (asterisk) unaffected by the EDIT pseudo instruction on the invocation statement option.
;	No	Comment line; (semicolon) unaffected by the EDIT pseudo instruction on the invocation statement option.
,	No	Continuation line; (comma) unaffected by the EDIT pseudo instruction on the invocation statement option (old format only).

Note: When the assembler edits "$CMNT", "$MIC", "$CNC", or "$APP", the string name and the pair of double quotation marks (" ") are replaced by a previously defined string. For example, when the assembler edits "$CMNT", a semicolon is substituted for the micro name $CMNT and the double quotation marks (" "). After the substitution occurs, the semicolon is not edited again and editing continues on the line. Using the predefined "$CMNT" micro permits a comment to be edited.

For example,

```
"$CMNT" Cray Inc  "$DATE" - "$TIME"
```

is edited as follows:

```
; Cray Inc  12/31/02 - 8:15:45
```

The characters to the right of the substituted character are shifted six positions to the left after editing, because the character string substituted for "$CMNT" (;) is six characters shorter than the micro name.

6.10.1 Micro Substitution

You can assign a micro name to a character string. You can refer to that character string in subsequent statements by its micro name. The assembler searches for quotation marks (") that delimit micro names. The first quotation mark indicates the beginning of a micro name; the second quotation mark identifies the end of a micro name. Before a statement is interpreted, the assembler replaces the micro name with the character string that comprises the micro. For more information on micros, see Section 6.7, page 88.

6.10.2 Concatenate

If statement editing is on, the concatenate feature combines characters connected by the underscore (_) character. the assembler examines each line for the underscore character and deletes each occurrence of the underscore. The two adjoining columns are linked before the statement is interpreted. The concatenate symbol can be in any column and tells the assembler to concatenate the characters following the last underscore to the character preceding the first underscore.

6.10.3 Append

The append feature combines source statements that continue for more than one line. It is available only when the new format is specified. The exact number of lines that the assembler can append depends on memory limitations.

The append symbol is a backslash character immediately followed by a newline character and appends one line to another. It can be used in any column on any line.

When the current line contains a blackslash character immediately followed by a newline character, the assembler appends the first non-whitespace character and all characters that follow from the next line to the current line. The characters are appended at the position in the current line that contains the backslash, the backslash and newline are replaced.

6.10.4 Continuation

A comma in column 1 indicates a continuation of the previous line. Columns 2 through 72 become a continuation of the previous line. Continuation is permitted only when the old format is specified.

6.10.5 Comment

A semicolon (;) in any column (new format only) or an asterisk (*) in column 1 indicates a comment line. The assembler lists comment lines, but they have no effect on the program. When a semicolon or an asterisk has an editing symbol after it, the symbol is treated as part of the comment and is not used. In the new format, comment statements with semicolons or asterisks are not appended.

Note: Asterisk comment statements are not included in macro definitions. To include a comment line in a macro definition, enter an underscore in column 1 of the comment line followed by an asterisk and then the comment. Because editing is disabled at definition time, the statement is inserted. If editing is enabled at expansion time, the underscore is edited out and the statement is treated as a comment.

The following example illustrates the use of comment statements in a macro:

```
      MACRO
      EXAMPLE
* This comment is not included in the definition.
_* This comment is included in the definition.
SYM    =   1
EXAMPLE  ENDM
```

The macro in the preceding example is expanded as follows:

```
      LIST          LIS,MAC
      EXAMPLE              ;Macro call
* This comment is included in the definition.
SYM    =   1
```

6.10.6 Actual Statements and Edited Statements

CAL statements can be divided into two categories: actual and edited. An *actual* statement is the unedited version of a statement that includes any appending of lines. It contains all of the editing symbols rather than the results of the editing. If an actual statement has a corresponding edited statement, further processing is done on the edited statement. The following examples show actual and edited statements.

This following example shows an actual statement:

```
LOC     MCALL           ARG1,\
                        ARG2,\
                        ARG3,\
                        ARG4,\
                        ARG5
```

An actual statement can have a corresponding edited statement. The *edited* statement displays the statement without any editing symbols. The following example shows the edited version of the actual statement in the preceding example:

```
LOC     MCALL           ARG1,ARG2,ARG3,ARG4,ARG5
```

In the following example, the actual statement has no corresponding edited statement:

```
    ENTER               ARG1,ARG2,ARG3        ; Comments
```

Cray X1 Instruction Set and Encoding [7]

This chapter lists the symbolic machine instructions and their respective binary encodings for Cray X1 systems. Some of the values in these tables are in binary; for instance, the j column in Section 7.6.4, page 115. See the *System Programmer Reference for Cray X1 Systems* for a complete description of each instruction.

This chapter is organized as follows:

7.1 Notation

The notations used in the instructions have the following meanings:

a*i*	Register *i* in the A registers.
{ }	Encloses optional elements in the syntax descriptions. The elements in the list are separated by the colon character. If the list ends with a colon with no element after the colon, that means that the syntax allows not specifying any element. For example, {,w:,l:} says there are three choices: ,w, l, or not specifying any element. And {s:d} says there are two choices: s or d.
c*k*	Register *k* in the control registers.
d	Double precision floating point value, 64 bits.
hnt	A *hint* in regards to making the best use of cache. For information on the meaning of the values

	when using atomic memory instructions, see Section 7.5.9.3, page 112. For the meanings of the values with vector memory instructions, see Section 7.7.10, page 126.
hw	Halfword value, 16 bits. The hw values for {a:b:c:d:ab:abc:abcd} are: a 00 b 01 c 10 d 11 ab 01 abc 10 abcd 11
imm6	Immediate; how wide (in bits) the field containing a constant value is. In the case of imm6, the field is 6 bits wide.
l	Longword value, 64 bits.
m*m*	Register *m* in the vector mask registers.
PC	Program counter. The program counter holds the address of the instruction currently executing.
s	Single precision floating point value, 32 bits.
sd	Single precision (32 bits) or double precision (64 bits) floating point value. The sd values for {s:d} are: s 00 d 01
s*j*	Register *j* in the S registers.
v*i*	Register *i* in the vector registers.
w	Word value, 32 bits.
wl	Word (32 bits) or longword (64 bits). The wl values for {,w:,l:} are: w 10 l 11
xxxxxx	Bits in this field are ignored by the hardware.
000000	Bits in this field must be zero.

In all symbolic machine instructions, alphabetic characters may be specified in uppercase or lowercase.

7.2 System and Memory Ordering Instructions

7.2.1 System Instructions

g	i	j	k	t	f	Syntax	Description
000000	xxxxxx	xxxxxx	xxxxxx	xx	000000	break {*n*}	Break. *n* is an integer of up to 20 bits that is placed in bits 25..6 of the instruction.
000001	xxxxxx	xxxxxx	xxxxxx	xx	*n*	syscall {*n*}	System call. *n*, an integer up to 6 bits, is the entry point number.

7.2.2 Memory Ordering Instructions

g	i	j	k	t	f	Syntax	Description
000010	000000	a*j*	000000	00	000000	gsync a*j*	Global ordering
000010	000000	a*j*	000000	00	000001	gsync a*j*,cpu	Global ordering and processor quiet
000010	000000	a*j*	000000	00	000010	gsync a*j*,a	Global ordering with respect to scalar loads
000010	000000	a*j*	000000	00	000011	gsync a*j*,r	Global ordering with respect to scalar stores
000010	000000	000000	000000	00	001000	lsync s,v	Local ordering of prior scalar with later vector
000010	000000	000000	000000	00	001001	lsync v,s	Local ordering of prior vector with later scalar
000010	000000	000000	000000	00	001010	lsync vr,s	Local ordering of prior vector reads with later scalar
000010	000000	000000	000000	00	001011	lsync v,v	Local ordering of vector references
000010	000000	v*j*	000000	00	001100	lsync v*j*,v	Local ordering of latest v*j* references with later vector

g	i	j	k	t	f	Syntax	Description
000010	000000	v*j*	000000	00	001101	lsync v*j*,v,el	Local elem ordering of latest v*j* references with later vector
000010	000000	000000	000000	00	001110	lsync i	Local ordering of stores and instruction fetches
000010	000000	000000	000000	00	001111	lsync fp	Local ordering of floating-point operations
000010	000000	a*j*	000000	00	010000	msync a*j*	Meta CPU synch
000010	000000	a*j*	000000	00	010001	msync a*j*,p	Meta CPU producer synch
000010	000000	a*j*	000000	00	010010	msync a*j*,v	Meta CPU vector synch
000010	000000	000000	000000	00	010100	mint	Meta CPU interrupt

7.3 Register Move Instructions

g	i	j	k	t	f	Syntax	Description
000011	a*i*	000000	s*k*	01	000000	a*i*{,d:,l:} s*k*	S to A move 64-bit
000011	a*i*	000000	s*k*	00	000000	a*i*,{s:w} s*k*	S to A move 32-bit
000011	s*i*	000000	a*k*	01	000001	s*i*{,d:,l:} a*k*	A to S move 64-bit
000011	s*i*	000000	a*k*	00	000001	s*i*,{s:w} a*k*	A to S move 32-bit
000011	a*i*	000000	c*k*	01	000010	a*i* c*k*	C to A move 64-bit
000011	c*i*	000000	a*k*	01	000011	c*i* a*k*	A to C move 64-bit
011100	000000	000000	a*k*	01	000000	vl a*k*	A to VL move
011100	a*i*	000000	000000	00	000001	a*i* vl	VL to A move
011100	a*i*	a*j*	v*k*	01	011100	a*i*{,d:,l:} v*k*,a*j*	V to A move 64-bit
011100	a*i*	a*j*	v*k*	00	011100	a*i*,{s:w} v*k*,a*j*	V to A move 32-bit
011100	s*i*	a*j*	v*k*	01	011000	s*i*{,d:,l:} v*k*,a*j*	V to S move 64-bit
011100	s*i*	a*j*	v*k*	00	011000	s*i*,{s:w} v*k*,a*j*	V to S move 32-bit
011100	a*i*	imm6	v*k*	01	011110	a*i*{,d:,l:} v*k*,imm6	V to A move 64-bit imm6
011100	a*i*	imm6	v*k*	00	011110	a*i*,{s:w} v*k*,imm6	V to A move 32-bit imm6

g	i	j	k	t	f	Syntax	Description
011100	s*i*	imm6	v*k*	01	011010	s*i*{,d:,l:} v*k*,imm6	V to S move 64-bit imm6
011100	s*i*	imm6	v*k*	00	011010	s*i*,{s:w} v*k*,imm6	V to S move 32-bit imm6
011100	v*i*	a*j*	a*k*	01	011101	v*i*,a*j* a*k*{,d:,l:}	A to V move 64-bit
011100	v*i*	a*j*	a*k*	00	011101	v*i*,a*j* s*k*,{s:w}	A to V move 32-bit
011100	v*i*	a*j*	s*k*	01	011001	v*i*,a*j* s*k*{,d:,l:}	S to V move 64-bit
011100	v*i*	a*j*	s*k*	00	011001	v*i*,a*j* s*k*,{s:w}	S to V move 32-bit
011100	v*i*	imm6	a*k*	01	011111	v*i*,imm6 a*k*{,d:,l:}	A to V move 64-bit imm6
011100	v*i*	imm6	a*k*	00	011111	v*i*,imm6 a*k*,{s:w}	A to V move 32-bit imm6
011100	v*i*	imm6	s*k*	01	011011	v*i*,imm6 s*k*{,d:,l:}	S to V move 64-bit imm6
011100	v*i*	imm6	s*k*	00	011100	v*i*,imm6 s*k*,{s:w}	S to V move 32-bit imm6

7.4 Jump Instructions

g	i	j	k	t	f	Syntax	Description
101111	a*i*	a*j*	000000	00	000000	j, a*i* a*j*	Jump a*i*<-PC+4;PC<-a*j*
101111	a*i*	a*j*	000000	00	000000	j, a*i* a*j*,sr	Jump subroutine a*i*<-PC+4;PC<-a*j*
101111	a*i*	a*j*	000000	00	000000	j, a*i* a*j*,rt	Jump return a*i*<-PC+4;PC<-a*j*

7.5 A Register Instructions

The following A register instructions are described in this section:

- A Register Integer Instructions (see Section 7.5.1, page 108).
- A Register Logical Instructions (see Section 7.5.2, page 108).
- A Register Shift Instructions (see Section 7.5.3, page 109).
- A Register Immediate Instructions (see Section 7.5.4, page 109).
- A Register Integer Compare Instructions (see Section 7.5.5, page 110).
- A Register Byte and Halfword Instructions (see Section 7.5.6, page 110).

- Other A Register Instructions (see Section 7.5.7, page 110).
- A Register Branch Instructions (see Section 7.5.8, page 111).
- A Register Memory Access Instructions (see Section 7.5.9, page 111).

7.5.1 A Register Integer Instructions

g	i	j	k	t	f	Syntax	Description
100010	a*i*	a*j*	a*k*	wl	000000	a*i*{,w:,l:} a*j*+a*k*	A int add
100011	a*i*	a*j*	imm8		000000	a*i*{,l:} a*j*+imm8	A int add long imm8
100011	a*i*	a*j*	imm8		101000	a*i*,w a*j*+imm8	A int add word imm8
100010	a*i*	a*j*	a*k*	wl	000101	a*i*{w:,l:} a*j*-s*k*	A int subtract
100011	a*i*	a*j*	imm8		000101	a*i*{,l:} a*j*-imm8	A int subtract long imm8
100011	a*i*	a*j*	imm8		101101	a*i*,w a*j*-imm8	A int subtract word imm8
100010	a*i*	a*j*	a*k*	wl	011000	a*i*{,w:,l:} a*j**a*k*	A int multiply
100010	a*i*	a*j*	a*k*	11	011001	a*i*{,l:} hi(a*j**a*k*)	A int multiply high
100010	a*i*	a*j*	a*k*	wl	001000	a*i*{,w:,l:} a*k*/a*j*	A int divide
100010	a*i*	a*j*	000000	wl	001010	a*i*{,w:,l:} lz(a*j*)	A leading zero count
100010	a*i*	a*j*	000000	wl	001011	a*i*{,w:,l:} pop(a*j*)	A population count

7.5.2 A Register Logical Instructions

g	i	j	k	t	f	Syntax	Description
100010	a*i*	a*j*	a*k*	11	010000	a*i*{,l:} a*j*&a*k*	A logical AND
100011	a*i*	a*j*	imm8		010000	a*i*{,l:} a*j*&imm8	A logical AND imm8
100010	a*i*	a*j*	a*k*	11	010001	a*i*{,l:} a*j*|a*k*	A logical OR
100011	a*i*	a*j*	imm8		010001	a*i*{,l:} a*j*|imm8	A logical OR imm8
100010	a*i*	a*j*	a*k*	11	010010	a*i*{,l:} ~a*j*^a*k*	A logical equivalence
100010	a*i*	a*j*	a*k*	11	010011	a*i*{,l:} a*j*^a*k*	A logical exclusive OR

g	i	j	k	t	f	Syntax	Description
100011	a*i*	a*j*	imm8		010011	a*i*{,l:} a*j*^imm8	A logical exclusive OR imm8
100010	a*i*	a*j*	a*k*	11	010100	a*i*{,l:} ~a*j*&a*k*	A logical ANDNOT

7.5.3 A Register Shift Instructions

g	i	j	k	f	t	Syntax	Description
100010	a*i*	a*j*	a*k*	wl	011100	a*i*{,w:,l:} a*j*<<a*k*	A shift left logical
100011	a*i*	a*j*	imm8		011100	a*i*{,l:} a*j*<<imm8	A shift left logical long imm8
100011	a*i*	a*j*	imm8		101110	a*i*,w a*j*<<imm8	A shift left logical word imm8
100010	a*i*	a*j*	a*k*	wl	011110	a*i*{,w:,l:} a*j*>>a*k*	A shift right logical
100011	a*i*	a*j*	imm8		011110	a*i*{,l:} a*j*>>imm8	A shift right logical long imm8
100011	a*i*	a*j*	imm8		101110	a*i*,w a*j*>>imm8	A shift right logical word imm8
100010	a*i*	a*j*	a*k*	11	011111	a*i*{,l:} +a*j*>>a*k*	A shift right arithmetic
100011	a*i*	a*j*	imm8		011111	a*i*{,l:} +a*j*>>imm8	A shift right arithmetic imm8

7.5.4 A Register Immediate Instructions

g	i	j	k	t	f	Syntax	Description
100000	a*i*	00hw	imm16			a*i*:{a:b:c:d} imm16	A insert imm16
100000	a*i*	01hw	imm16			a*i* imm16:{a:b:c:d}	A clear and enter imm16
100000	a*i*	10hw	imm16			a*i*:{ab:abc} imm16	A insert sign_ext(imm16)
100000	a*i*	11hw	imm16			a*i* imm16:{ab:abc:abcd}	A clear and enter sign_ext(imm16)

7.5.5 A Register Integer Compare Instructions

g	i	j	k	t	f	Syntax	Description
100010	a*i*	a*j*	a*k*	11	110010	a*i*{,l:} a*j*< a*k*	A int compare less than
100011	a*i*	a*j*	imm8		110010	a*i*{,l:} a*j*<imm8	A int compare less than imm8
100010	a*i*	a*j*	a*k*	11	111010	a*i*,u{l:} a*j*<a*k*	A int compare less than, unsigned
100011	a*i*	a*j*	imm8		111010	a*i*,u{l:} a*j*<imm8	A int compare less than unsigned imm8

7.5.6 A Register Byte and Halfword Instructions

g	i	j	k	t	f	Syntax	Description
000101	a*i*	a*j*	a*k*	11	100000	a*i*{,l:} extb(a*j*, a*k*)	A extract byte
000101	a*i*	a*j*	a*k*	11	100001	a*i*{,l:} exth(a*j*, a*k*)	A extract halfword
000101	a*i*	a*j*	a*k*	11	100100	a*i*,{l:} insb(a*j*, a*k*)	A insert byte
000101	a*i*	a*j*	a*k*	11	100101	a*i*,{l:} insh(a*j*, a*k*)	A insert halfword
000101	a*i*	a*j*	a*k*	11	100010	a*i*,{l:} mskb(a*j*, a*k*)	A mask byte
000101	a*i*	a*j*	a*k*	11	100011	a*i*,{l:} mskh(a*j*, a*k*)	A mask halfword

7.5.7 Other A Register Instructions

g	i	j	k	t	f	Syntax	Description
100010	a*i*	a*j*	a*k*	11	010111	a*i*{,l:} a*j*?a*i*:a*k*	A move if zero
100010	a*i*	a*j*	a*k*	11	010110	a*i*{,l:} a*j*?a*k*:a*i*	A move if nonzero
100010	a*i*	000000	a*k*	11	000111	a*i*{,l:} cvl(a*k*)	A compute vector length
100011	a*i*	000000	imm8		000111	a*i*{,l:} cvl(imm8)	A compute vector length imm8

7.5.8 A Register Branch Instructions

g	i	k	Syntax	Description
101000	a*i*	imm20	bz a*i*,imm20	A branch (a*i*==0) PC<-PC+4+(imm20<<2)
101001	a*i*	imm20	bn a*i*,imm20	A branch (a*i*!=0) PC<-PC+4+(imm20<<2)
101010	a*i*	imm20	ble a*i*,imm20	A branch (a*i*<=0) PC<-PC+4+(imm20<<2)
101011	a*i*	imm20	blt a*i*,imm20	A branch (a*i*<0) PC<-PC+4+(imm20<<2)
101100	a*i*	imm20	bge a*i*,imm20	A branch (a*i*>=0) PC<-PC+4+(imm20<<2)
101101	a*i*	imm20	bgt a*i*,imm20	A branch (a*i*>0) PC<-PC+4+(imm20<<2)

Note: For A register and S register branch instructions.

In the branch instructions, the imm20 field is essentially an instruction offset (relative to the instruction immediately after the branch instruction). If the expression in the imm20 field of the symbolic instruction has address attribute (e.g., a relocatable symbol), then the assembler divides the value of the expression by 4 to convert to an instruction offset (4 bytes per instruction) and stores the value in the imm20 field of the machine instruction. If the expression in the imm20 field of the symbolic instruction has absolute attribute (e.g., a constant integer), then the assembler does no conversion and stores the value of the expression in the imm20 field of the machine instruction.

7.5.9 A Register Memory Access Instructions

The following A register memory access instructions are described in this section:

- Load and Store Instructions (see Section 7.5.9.1, page 111).
- Prefetch Instructions (see Section 7.5.9.2, page 112).
- Atomic Memory Instructions (see Section 7.5.9.3, page 112).

7.5.9.1 Load and Store Instructions

g	i	j	k	t	f	Syntax	Description
100101	a*i*	a*j*	imm14			a*i*{,d:,l:} [a*j*+imm14]	A load 64-bit imm14

g	i	j	k	t	f	Syntax	Description
100100	a*i*	a*j*	imm14			a*i*,{s:w} [a*j*+imm14]	A load 32-bit, sign-extended imm14
100001	a*i*	a*j*	a*k*	01	000000	a*i*{,d:,l:} [a*j*+a*k*]	A load 64-bit indexed
100001	a*i*	a*j*	a*k*	00	000000	a*i*,{s:w} [a*j*+a*k*]	A load 32-bit sign-extended indexed
100111	a*i*	a*j*	imm14			[a*j*+imm14] a*i*{,d:,l:}	A store 64-bit imm14
100110	a*i*	a*j*	imm14			[a*j*+imm14] a*i*,{s:w}	A store 32-bit imm14
100001	a*i*	a*j*	a*k*	01	000001	[a*j*+a*k*] a*i*{,d:,l:}	A store 64-bit indexed
100001	a*i*	a*j*	a*k*	00	000001	[a*j*+a*k*] a*i*,{s:w}	A store 32-bit indexed
100001	a*i*	a*j*	000000	01	000010	a*i*{,d:,l:} [a*j*],ua	A load 64-bit unaligned
100001	a*i*	a*j*	000000	01	000011	[a*j*] a*i*{,d:,l:},ua	A store 64-bit unaligned

7.5.9.2 Prefetch Instructions

g	i	j	k	t	f	Syntax	Description
100101	000000	a*j*	imm14			pref {,d:,l:}[a*j*+imm14]	Prefetch 64-bit imm14
100001	000000	a*j*	a*k*	01	000000	pref {,d:,l:}[a*j*+a*k*]	Prefetch 64-bit indexed
100100	000000	a*j*	imm14			pref,{s:w} [a*j*+imm14]	Prefetch 32-bit sign-extended imm14
100001	000000	a*j*	a*k*	00	000000	pref,{s:w} [a*j*+a*k*]	Prefetch 32-bit sign-extended indexed

7.5.9.3 Atomic Memory Instructions

In the following instructions, the *hnt* (hints) fields advise the hardware on the anticipated access patterns for the referenced memory location. They have the following meanings:

ex (000) Exclusive. Read misses should allocate the line exclusively, anticipating a subsequent write by the same MSP. Write misses should allocate the line (and must do so exclusively).

na (010) Non-allocate. Read and write references should not allocate space for the line in the local cache. This hint should be used

when no temporal locality is expected (to avoid cache pollution), or the reference is to memory that a different MSP will access subsequently (explicit communication). If the referenced item is present in the cache, the cache should be accessed normally (this is not a cache bypass mode). If no hint is specified, the hint defaults to na.

g	i	j	k	t	f	Syntax	Description
000100	a*i*	a*j*	a*k*	01	*hnt*000	a*i*{,d:,l:} [a*j*],afadd,a*k*{,hnt}	Atomic fetch and add
000100	a*i*	a*j*	a*k*	01	*hnt*001	a*i*{,d:,l:} [a*j*],afax,a*k*{,hnt}	Atomic fetch and exclusive OR
000100	a*i*	a*j*	a*k*	01	*hnt*010	a*i*{,d:,l:} [a*j*],acswap,a*k*{,hnt}	Atomic compare and swap
000100	a*i*	a*j*	000000	01	*hnt*100	[a*j*] a*i*{,d:,l:},aadd{,hnt}	Atomic add
000100	a*i*	a*j*	a*k*	01	*hnt*101	[a*j*] a*i*{,d:,l:},aax,ak{,hnt}	Atomic and exclusive OR

7.6 S Register Instructions

The following S register instructions are described in this section:

- S Register Integer Instructions (see Section 7.6.1, page 114).
- S Register Logical Instructions (see Section 7.6.2, page 114).
- S Register Shift Instructions (see Section 7.6.3, page 115).
- S Register Immediate Instructions (see Section 7.6.4, page 115).
- S Register Integer Compare Instructions (see Section 7.6.5, page 115).
- Other S Register Instructions (see Section 7.6.6, page 116).
- S Register Branch Instructions (see Section 7.6.7, page 116).
- S Register Floating Point Instructions (see Section 7.6.8, page 117).
- S Register Floating Point Compare Instructions (see Section 7.6.9, page 117).
- S Register Conversion Instructions (see Section 7.6.10, page 118).
- S Register Memory Instructions (see Section 7.6.11, page 119).

7.6.1 S Register Integer Instructions

g	i	j	k	t	f	Syntax	Description
110010	*si*	*sj*	*sk*	wl	000000	*si*{,w:,l:} *sj*+*sk*	S int add
110011	*si*	*sj*	imm8		000000	*si*{,l:} *sj*+imm8	S int add long imm8
110011	*si*	*sj*	imm8		101000	*si*,w *sj*+imm8	S int add word imm8
110010	*si*	*sj*	*sk*	wl	000101	*si*{,w:,l:} *sj*-*sk*	S int subtract
110011	*si*	*sj*	imm8		000101	*si*{,l:} *sj*-imm8	S int subtract long imm8
110011	*si*	*sj*	imm8		101101	*si*,w *sj*-imm8	S int subtract word imm8
110010	*si*	*sj*	*sk*	wl	011000	*si*{,w:,l:} *sj***sk*	S int multiply
110010	*si*	*sj*	*sk*	11	011001	*si*{,l:} hi(*sj***sk*)	S int multiply high
110010	*si*	*sj*	*sk*	wl	001000	*si*{,w:,l:} *sk*/*sj*	S int divide
110010	*si*	*sj*	000000	wl	001010	*si*{,w:,l:} lz(*sj*)	S leading zero count
110010	*si*	*sj*	000000	wl	001011	*si*{,w:,l:} pop(*sj*)	S population count

7.6.2 S Register Logical Instructions

g	i	j	k	t	f	Syntax	Description
110010	*si*	*sj*	*sk*	11	010000	*si*{,l:} *sj*&*sk*	S logical AND
110011	*si*	*sj*	imm8		010000	*si*{,l:} *sj*&imm8	S logical AND imm8
110010	*si*	*sj*	*sk*	11	010001	*si*{,l:} *sj*|*sk*	S logical OR
110011	*si*	*sj*	imm8		010001	*si*{,l:} *sj*|imm8	S logical OR imm8
110010	*si*	*sj*	*sk*	11	010010	*si*{,l:} ~*sj*^*sk*	S logical equivalence
110010	*si*	*sj*	*sk*	11	010011	*si*{,l:} *sj*^*sk*	S logical exclusive OR
110011	*si*	*sj*	imm8		010011	*si*{,l:} *sj*^imm8	S logical exclusive OR imm8
110010	*si*	*sj*	*sk*	11	010100	*si*{,l:} ~*sj*&*sk*	S logical ANDNOT

7.6.3 S Register Shift Instructions

g	i	j	k	t	f	Syntax	Description
110010	*si*	*sj*	*sk*	wl	011100	*si*{,w:,l:} *sj*<<*sk*	S shift left logical
110011	*si*	*sj*	imm8		011100	*si*{,l:} *sj*<<imm8	S shift left logical long imm8
110011	*si*	*sj*	imm8		101100	*si*,w *sj*<<imm8	S shift left logical word imm8
110010	*si*	*sj*	*sk*	wl	011110	*si*{,w:,l:} *sj*>>*sk*	S shift right logical
110011	*si*	*sj*	imm8		011110	*si*{,l:} *sj*>>imm8	S shift right logical long imm8
110011	*si*	*sj*	imm8		101110	*si*,w *sj*>>imm8	S shift right logical word imm8
110010	*si*	*sj*	*sk*	11	011111	*si*{,l:} +*sj*>>*sk*	S shift right arithmetic
110011	*si*	*sj*	imm8		011111	*si*{,l:} +*sj*>>imm8	S shift right arithmetic imm8

7.6.4 S Register Immediate Instructions

g	i	j	k	t	f	Syntax	Description
110000	*si*	00hw	imm16			*si*:{a:b:c:d} imm16	S insert imm16
110000	*si*	01hw	imm16			*si* imm16:{a:b:c:d}	S clear and enter imm16
110000	*si*	10hw	imm16			*si*:{ab:abc} imm16	S insert sign_ext(imm16)
110000	*si*	11hw	imm16			*si* imm16:{ab:abc:abcd}	S clear and enter sign_ext(imm16)

7.6.5 S Register Integer Compare Instructions

g	i	j	k	t	f	Syntax	Description
110010	*si*	*sj*	*sk*	11	110010	*si*{,l:} *sj*< *sk*	S int compare less than
110011	*si*	*sj*	imm8		110010	*si*{,l:} *sj*<imm8	S int compare less than imm8
110010	*si*	*sj*	*sk*	11	111010	*si*,u{l:} *sj*<*sk*	S int compare less than unsigned
110011	*si*	*sj*	imm8		111010	*si*,u{l:} *sj*<imm8	S int compare less than unsigned imm8

7.6.6 Other S Register Instructions

g	i	j	k	t	f	Syntax	Description
110010	s*i*	s*j*	s*k*	11	010111	s*i*{,l:} s*j*?s*i*:s*k*	S move if zero
110010	s*i*	s*j*	s*k*	11	010110	s*i*{,l:} s*j*?s*k*:s*i*	S move if nonzero
110010	s*i*	s*j*	s*k*	01	001100	s*i*{,d:,l:} cpys(s*j*,s*k*)	S copy sign 64-bit
110010	s*i*	s*j*	s*k*	00	001100	s*i*{,s:w} cpys(s*j*,s*k*)	S copy sign 32-bit
110010	s*i*	000000	s*k*	11	101111	s*i* bmm(s*k*)	S bit matrix multiply

7.6.7 S Register Branch Instructions

g	i	j	k	t	f	Syntax	Description
111000	s*i*		imm20			bz s*i*,imm20	S branch (s*i*==0) PC<-PC+4+(imm20<<2)
111001	s*i*		imm20			bn s*i*,imm20	S branch (s*i*=0) PC<-PC+4+(imm20<<2)
111010	s*i*		imm20			ble s*i*,imm20	S branch (s*i*<=0) PC<-PC+4+(imm20<<2)
111011	s*i*		imm20			blt s*i*,imm20	S branch (s*i*<0) PC<-PC+4+(imm20<<2)
111100	s*i*		imm20			bge s*i*,imm20	S branch (s*i*>=0) PC<-PC+4+(imm20<<2)
111101	s*i*		imm20			bgt s*i*,imm20	S branch (s*i*>0) PC<-PC+4+(imm20<<2)

Note: For A register and S register branch instructions.

In the branch instructions, the imm20 field is essentially an instruction offset (relative to the instruction immediately after the branch instruction). If the expression in the imm20 field of the symbolic instruction has address attribute (e.g., a relocatable symbol), then the assembler divides the value of the expression by 4 to convert to an instruction offset (4 bytes per instruction) and stores the value in the imm20 field of the machine instruction. If the expression in the imm20 field of the symbolic instruction has absolute attribute (e.g., a constant integer), then the assembler does no conversion and stores the value of the expression in the imm20 field of the machine instruction.

7.6.8 S Register Floating Point Instructions

g	i	j	k	t	f	Syntax	Description
110010	*si*	*sj*	*sk*	sd	000000	*si*,{s:d} *sj*+*sk*	S floating point add
110010	*si*	*sj*	*sk*	sd	000101	*si*,{s:d} *sj*-*sk*	S floating point subtract
110010	*si*	*sj*	*sk*	sd	011000	*si*,{s:d} *sj***sk*	S floating point multiply
110010	*si*	*sj*	*sk*	sd	001000	*si*,{s:d} *sj*/*sk*	S floating point divide
110010	*si*	*sj*	000000	sd	001001	*si*,{s:d} sqrt(*sj*)	S floating point square root
110010	*si*	*sj*	000000	sd	001111	*si*,{s:d} abs(*sj*)	S floating point absolute value

7.6.9 S Register Floating Point Compare Instructions

g	i	j	k	t	f	Syntax	Description
110010	*si*	*sj*	*sk*	sd	110001	*si*,{s:d} *sj*==*sk*	S floating point compare equal
110010	*si*	*sj*	*sk*	sd	111110	*si*,{s:d} *sj*!=*sk*	S floating point compare not equal
110010	*si*	*sj*	*sk*	sd	110010	*si*,{s:d} *sj*<*sk*	S floating point compare less than
110010	*si*	*sj*	*sk*	sd	110011	*si*,{s:d} *sj*<=*sk*	S floating point compare less than or equal
110010	*si*	*sj*	*sk*	sd	110000	*si*,{s:d} *sj*?*sk*	S floating point compare unordered

7.6.10 S Register Conversion Instructions

g	i	j	k	t	f	Syntax	Description
110010	*si*	*sj*	000000	01	101010	*si*,d *sj*,w	S convert word to double
110010	*si*	*sj*	000000	01	101011	*si*,d *sj*,l	S convert long to double
110010	*si*	*sj*	000000	01	101000	*si*,d *sj*,s	S convert single to double
110010	*si*	*sj*	000000	00	101010	*si*,s *sj*,w	S convert word to single
110010	*si*	*sj*	000000	00	101011	*si*,s *sj*,l	S convert long to single
110010	*si*	*sj*	000000	00	101001	*si*,s *sj*,d	S convert double to single
110010	*si*	*sj*	000000	11	101000	*si*,l *sj*,s	S convert single to long
110010	*si*	*sj*	000000	11	101001	*si*,l *sj*,d	S convert double to long
110010	*si*	*sj*	000000	10	101000	*si*,w *sj*,s	S convert single to word
110010	*si*	*sj*	000000	10	101001	*si*,w *sj*,d	S convert double to word
110010	*si*	*sj*	000000	11	100000	*si*,l round(*sj*),s	S round single to long
110010	*si*	*sj*	000000	11	100001	*si*,l round(*sj*),d	S round double to long
110010	*si*	*sj*	000000	10	100000	*si*,w round(*sj*),s	S round single to word
110010	*si*	*sj*	000000	10	100001	*si*,w round(*sj*),d	S round double to word
110010	*si*	*sj*	000000	11	100010	*si*,l trunc(*sj*),s	S truncate single to long
110010	*si*	*sj*	000000	11	100011	*si*,l trunc(*sj*),d	S truncate double to long
110010	*si*	*sj*	000000	10	1000100	*si*,w trunc(*sj*),s	S truncate single to word
110010	*si*	*sj*	000000	10	100011	*si*,w trunc(*sj*),d	S truncate double to word
110010	*si*	*sj*	000000	11	100100	*si*,l ceil(*sj*),s	S ceiling single to long
110010	*si*	*sj*	000000	11	100101	*si*,l ceil(*sj*),d	S ceiling double to long
110010	*si*	*sj*	000000	10	100100	*si*,w ceil(*sj*),s	S ceiling single to word
110010	*si*	*sj*	000000	10	100101	*si*,w ceil(*sj*),d	S ceiling double to word
110010	*si*	*sj*	000000	11	100110	*si*,l floor(*sj*),s	S floor single to long
110010	*si*	*sj*	000000	11	100111	*si*,l floor(*sj*),d	S floor double to long
110010	*si*	*sj*	000000	10	100110	*si*,w floor(*sj*),s	S floor single to word
110010	*si*	*sj*	000000	10	100111	*si*,w floor(*sj*),d	S floor double to word

7.6.11 S Register Memory Instructions

g	i	j	k	t	f	Syntax	Description
110101	*si*	a*j*	imm14			*si*{,d:,l:} [a*j*+imm14]	S load 64-bit imm14
110100	*si*	a*j*	imm14			*si*,{s:w} [a*j*+imm14]	S load 32-bit sign_extend imm14
110001	*si*	a*j*	a*k*	01	000000	*si*{,d:,l:} [a*j*+a*k*]	S load 64-bit indexed
110001	*si*	a*j*	a*k*	00	000000	*si*,{s:w} [a*j*+a*k*]	S load 32-bit sign_extend indexed
110111	*si*	a*j*	imm14			[a*j*+imm14] *si*{,d:,l:}	S store 64-bit imm14
110110	*si*	a*j*	imm14			[a*j*+imm14] *si*,{s:w}	S store 32-bit imm14
110001	*si*	a*j*	a*k*	01	000001	[a*j*+a*k*] *si*{,d:,l:}	S store 64-bit indexed
110001	*si*	a*j*	a*k*	00	000001	[a*j*+a*k*] *si*,{s:w}	S store 32-bit indexed

7.7 Vector Register Instructions

The instructions in the following sections use the vector registers:

- Vector Register Integer Instructions (see Section 7.7.1, page 120).
- Vector Register Logical Instructions (see Section 7.7.2, page 120).
- Vector Register Shift Instructions (see Section 7.7.3, page 121).
- Vector Register Integer Compare Instructions (see Section 7.7.4, page 121).
- Vector Register Floating Point Instructions (see Section 7.7.5, page 122).
- Vector Register Floating Point Compare Instructions (see Section 7.7.6, page 123).
- Vector Register Conversion Instructions (see Section 7.7.7, page 123).
- Other Vector Register Instructions (see Section 7.7.8, page 125).
- Vector Mask Instructions (see Section 7.7.9, page 125).
- Vector Memory Instructions (see Section 7.7.10, page 126).
- Privileged Instructions (see Section 7.7.11, page 127).

The notation mm in the vector syntax refers to one of the vector mask registers.

7.7.1 Vector Register Integer Instructions

g	i	j	k	t	f	Syntax	Description
0100mm	vi	vj	vk	wl	000000	vi{,w:,l:} vj+vk,mm	V int add
0101mm	vi	vj	sk	wl	000000	vi,{,w:,l:} vj+sk,mm	V/S int add
0100mm	vi	vj	vk	11	000010	vi,vc vj+vk,mm	V int add with carry
0101mm	vi	vj	sk	11	000010	vi,vc vj+sk,mm	V/S int add with carry
0100mm	vi	vj	vk	wl	000001	vi{,w:,l:} vk-vj,mm	V int subtract
0101mm	vi	vj	sk	wl	000001	vi{,w:,l:} sk-vj,mm	V/S int subtract
0100mm	vi	vj	vk	11	000011	vi, vc vk-vj,mm	V int subtract with borrow
0101mm	vi	vj	sk	11	000011	vi,cv sk-vj,mm	V/S int subtract with borrow
0100mm	vi	vj	vk	wl	011000	vi{,w:,l:} vj*vk,mm	V int multiply
0101mm	vi	vj	sk	wl	011000	vi{,w:,l:} vj*sk,mm	V/S int multiply
0100mm	vi	vj	vk	11	011001	vi{,l:} hi(vj*vk),mm	V int multiply high
0101mm	vi	vj	sk	11	011001	vi{,l:} hi(vj*sk),mm	V/S int multiply high
0100mm	vi	vj	000000	wl	001010	vi{,w:,l:} lz(vj),mm	V leading zero count
0100mm	vi	vj	000000	wl	001011	vi{,w:,l:} pop(vj),mm	V population count

7.7.2 Vector Register Logical Instructions

g	i	j	k	t	f	Syntax	Description
0100mm	vi	vj	vk	wl	010000	vi{,w:,l:} vj&vk,mm	V logical AND
0101mm	vi	vj	sk	wl	010000	vi{,w:,l:} vj&sk,mm	V/S logical AND
0100mm	vi	vj	vk	wl	010001	vi{,w:,l:} vj|vk,mm	V logical OR
0101mm	vi	vj	sk	wl	010001	vi{,w:,l:} vj|sk,mm	V/S logical OR
0100mm	vi	vj	vk	wl	010010	vi{,w:,l:} ~vj^vk,mm	V logical equivalence

0101m*m*	v*i*	v*j*	s*k*	wl	010010	v*i*{,w:,l:} ~v*j*^s*k*,m*m*	V/S logical equivalence
0100m*m*	v*i*	v*j*	v*k*	wl	010011	v*i*{,w:,l:} v*j*^v*k*,m*m*	V logical exclusive OR
0101m*m*	v*i*	v*j*	s*k*	wl	010011	v*i*{,w:,l:} v*j*^s*k*,m*m*	V/S logical exclusive OR
0100m*m*	v*i*	v*j*	v*k*	wl	010100	v*i*{,w:,l:} ~v*j*&v*k*,m*m*	V logical ANDNOT
0101m*m*	v*i*	v*j*	s*k*	wl	010100	v*i*{,w:,l:} ~v*j*&s*k*,m*m*	V/S logical ANDNOT

7.7.3 Vector Register Shift Instructions

g	i	j	k	t	f	Syntax	Description
0100m*m*	v*i*	v*j*	v*k*	wl	011100	v*i*{,w:,l:} v*j*<<v*k*,m*m*	V shift left logical
0101m*m*	v*i*	v*j*	s*k*	wl	011100	v*i*{,w:,l:} v*j*<<s*k*,m*m*	V/S shift left logical
0100m*m*	v*i*	v*j*	v*k*	wl	011110	v*i*{,w:,l:} v*j*>>v*k*,m*m*	V shift right logical
0101m*m*	v*i*	v*j*	s*k*	wl	011110	v*i*{,w:,l:} v*j*>>sk,m*m*	V/S shift right logical
0100m*m*	v*i*	v*j*	v*k*	wl	011111	v*i*{,w:,l:} +v*j*>>v*k*,m*m*	V shift right arithmetic
0101m*m*	v*i*	v*j*	s*k*	wl	011111	v*i*{,w:,l:} +v*j*>>s*k*,m*m*	V/S shift right arithmetic

7.7.4 Vector Register Integer Compare Instructions

g	i	j	k	t	f	Syntax	Description
0100m*m*	000m*i*	v*j*	v*k*	wl	110001	m*i*{,w:,l:} v*j*== v*k*,m*m*	V int compare equal
0101m*m*	000m*i*	v*j*	s*k*	wl	110001	m*i*{,w:,l:} v*j*== s*k*,m*m*	V/S int compare equal
0100m*m*	000m*i*	v*j*	v*k*	wl	111110	m*i*{,w:,l:} v*j*!=v*k*,m*m*	V int compare not equal
0101m*m*	000m*i*	v*j*	s*k*	wl	111110	m*i*{,w:,l:} v*j*!= s*k*,m*m*	V/S int compare not equal
0100m*m*	000m*i*	v*j*	v*k*	wl	110010	m*i*{,w:,l:} v*j*<v*k*,m*m*	V int compare less than
0101m*m*	000m*i*	v*j*	s*k*	wl	110010	m*i*{,w:,l:} v*j*<s*k*,m*m*	V/S int compare less than
0100m*m*	000m*i*	v*j*	v*k*	wl	110010	m*i*,u{,w:,l:} v*j*<v*k*,m*m*	V int compare less than unsigned
0101m*m*	000m*i*	v*j*	s*k*	wl	110010	m*i*,u{,w:,l:} v*j*<s*k*,m*m*	V/S int compare less than unsigned

g	i	j	k	t	f	Syntax	Description
0100m*m*	000m*i*	v*j*	v*k*	wl	111011	m*i*{,w:,l:} v*j*<= v*k*,m*m*	V int compare less than or equal
0101m*m*	000m*i*	v*j*	s*k*	wl	111011	m*i*{,w:,l:} v*j*<= s*k*,m*m*	V/S int compare less than or equal
0100m*m*	000m*i*	v*j*	v*k*	wl	111011	m*i*,u{,w:,l:} v*j*<= v*k*,m*m*	V int compare less than or equal unsigned
0101m*m*	000m*i*	v*j*	s*k*	wl	111011	m*i*,u{,w:,l:} v*j*<= s*k*,m*m*	V/S int compare less than or equal unsigned
0101m*m*	000m*i*	v*j*	s*k*	wl	110100	m*i*{,w:,l:} v*j*>s*k*,m*m*	V/S int compare greater than
0101m*m*	000m*i*	v*j*	s*k*	wl	111100	m*i*,u{,w:,l:} v*j*> s*k*,m*m*	V/S int compare greater than unsigned
0101m*m*	000m*i*	v*j*	s*k*	wl	110101	m*i*{,w:,l:} v*j*>= s*k*,m*m*	V/S int compare greater than or equal
0101m*m*	000m*i*	v*j*	s*k*	wl	111101	m*i*,u{,w:,l:} v*j*>= s*k*,m*m*	V/S int compare greater than or equal unsigned

7.7.5 Vector Register Floating Point Instructions

g	i	j	k	t	f	Syntax	Description
0100m*m*	v*i*	v*j*	v*k*	sd	000000	v*i*,{s:d} v*j*+s*k*,m*m*	V floating point add
0101m*m*	v*i*	v*j*	s*k*	sd	000000	v*i*,{s:d} v*j*+v*k*,m*m*	V/S floating point add
0100m*m*	v*i*	v*j*	v*k*	sd	000001	v*i*,{s:d} v*k*-v*j*,m*m*	V floating point subtract
0101m*m*	v*i*	v*j*	s*k*	sd	000001	v*i*,{s:d} s*k*-v*j*,m*m*	V/S floating point subtract
0100*m*	v*i*	v*j*	v*k*	sd	011000	v*i*,{s:d} v*j**v*k*,m*m*	V floating point multiply
0101m*m*	v*i*	v*j*	v*k*	sd	011000	v*i*,{s:d} v*j**s*k*,m*m*	V/S floating point multiply
0100m*m*	v*i*	v*j*	v*k*	sd	001000	v*i*,{s:d} v*k*/v*j*,m*m*	V floating point divide
0101m*m*	v*i*	v*j*	s*k*	sd	001000	v*i*,{s:d} s*k*/v*j*,m*m*	V/S floating point divide
0100m*m*	v*i*	v*j*	000000	sd	001001	v*i*,{s:d} sqrt(v*j*),m*m*	V floating point square root
0100m*m*	v*i*	v*j*	000000	sd	001111	v*i*,{s:d} abs(v*j*),m*m*	V floating point absolute value

7.7.6 Vector Register Floating Point Compare Instructions

g	i	j	k	t	f	Syntax	Description
0100m*m*	00m*i*	v*j*	v*k*	sd	110001	m*i*,{s:d} v*j*==v*k*,m*m*	V floating point compare equal
0101m*m*	00m*i*	v*j*	s*k*	sd	110001	m*i*,{s:d} v*j*==s*k*,m*m*	V/S floating point compare equal
0100m*m*	00m*i*	v*j*	v*k*	sd	111110	m*i*,{s:d} v*j*!=v*k*,m*m*	V floating point compare not equal
0101m*m*	00m*i*	v*j*	s*k*	sd	111110	m*i*,{s:d} v*j*!=s*k*,m*m*	V/S floating point compare not equal
0100m*m*	00m*i*	v*j*	v*k*	sd	110010	m*i*,{s:d} v*j*<v*k*,m*m*	V floating point compare less than
0101m*m*	00m*i*	v*j*	s*k*	sd	110010	m*i*,{s:d} v*j*<s*k*,m*m*	V/S floating point compare less than
0100m*m*	00m*i*	v*j*	v*k*	sd	110011	m*i*,{s:d} v*j*<=v*k*,m*m*	V floating point compare less than or equal
0101m*m*	00m*i*	v*j*	s*k*	sd	110010	m*i*,{s:d} v*j*<=s*k*,m*m*	V/S floating point compare less than or equal
0101m*m*	00m*i*	v*j*	s*k*	sd	110100	m*i*,{s:d} v*j*>s*k*,m*m*	V/S floating point compare greater than
0101m*m*	00m*i*	v*j*	s*k*	sd	110101	m*i*,{s:d} v*j*>=s*k*,m*m*	V/S floating point compare greater than or equal
0100m*m*	00m*i*	v*j*	v*k*	sd	111000	m*i*,{s:d} v*j*?v*k*,m*m*	V floating point compare unordered
0101m*m*	00m*i*	v*j*	s*k*	sd	111000	m*i*,{s:d} v*j*?s*k*,m*m*	V/S floating point compare unordered

7.7.7 Vector Register Conversion Instructions

g	i	j	k	t	f	Syntax	Description
0100m*m*	v*i*	v*j*	000000	01	101010	v*i*,d v*j*,w,m*m*	V convert word to double
0100m*m*	v*i*	v*j*	000000	01	101011	v*i*,d v*j*,l,m*m*	V convert long to double

g	i	j	k	t	f	Syntax	Description
0100mm	v*i*	v*j*	000000	01	101000	v*i*,d v*j*,s,m*m*	V convert single to double
0100mm	v*i*	v*j*	000000	00	101010	v*i*,s v*j*,w,m*m*	V convert word to single
0100mm	v*i*	v*j*	000000	00	101011	v*i*,s v*j*,l,m*m*	V convert long to single
0100mm	v*i*	v*j*	000000	00	101001	v*i*,s v*j*,d,m*m*	V convert double to single
0100mm	v*i*	v*j*	000000	11	101010	v*i*,l v*j*,w,m*m*	V convert word to long
0100mm	v*i*	v*j*	000000	11	101000	v*i*,l v*j*,s,m*m*	V convert single to long
0100mm	v*i*	v*j*	000000	11	101001	v*i*,l v*j*,d,m*m*	V convert double to long
0100mm	v*i*	v*j*	000000	10	101011	v*i*,w v*j*,l,m*m*	V convert long to word
0100mm	v*i*	v*j*	000000	10	101000	v*i*,w v*j*,s,m*m*	V convert single to word
0100mm	v*i*	v*j*	000000	10	101001	v*i*,w v*j*,d,m*m*	V convert double to word
0100mm	v*i*	v*j*	000000	11	101001	v*i*,l round(v*j*),s,m*m*	V round single to long
0100mm	v*i*	v*j*	000000	11	100001	v*i*,l round(v*j*),d,m*m*	V round double to long
0100mm	v*i*	v*j*	000000	10	100000	v*i*,w round(v*j*),s,m*m*	V round single to word
0100mm	v*i*	v*j*	000000	10	100001	v*i*,w round(v*j*),d,m*m*	V round double to word
0100mm	v*i*	v*j*	000000	11	100010	v*i*,l trunc(v*j*),s,m*m*	V truncate single to long
0100mm	v*i*	v*j*	000000	11	100011	v*i*,l trunc(v*j*),d,m*m*	V truncate double to long
0100mm	v*i*	v*j*	000000	10	100010	v*i*,w trunc(v*j*),s,m*m*	V truncate single to word
0100mm	v*i*	v*j*	000000	10	100011	v*i*,w trunc(v*j*),d,m*m*	V truncate double to word
0100mm	v*i*	v*j*	000000	11	100100	v*i*,l ceil(v*j*),s,m*m*	V ceiling single to long
0100mm	v*i*	v*j*	000000	11	100101	v*i*,l ceil(v*j*),d,m*m*	V ceiling double to long
0100mm	v*i*	v*j*	000000	10	100100	v*i*,w ceil(v*j*),s,m*m*	V ceiling single to word
0100mm	v*i*	v*j*	000000	10	100101	v*i*,w ceil(v*j*),d,m*m*	V ceiling double to word
0100mm	v*i*	v*j*	000000	11	100110	v*i*,l floor(v*j*),s,m*m*	V floor single to long
0100mm	v*i*	v*j*	000000	11	100111	v*i*,l floor(v*j*),d,m*m*	V floor double to long
0100mm	v*i*	v*j*	000000	10	100110	v*i*,w floor(v*j*),s,m*m*	V floor single to word
0100mm	v*i*	v*j*	000000	10	100111	v*i*,w floor(v*j*),d,m*m*	V floor double to word

7.7.8 Other Vector Register Instructions

g	i	j	k	t	f	Syntax	Description
011100	000000	000000	v*k*	11	000010	bmm v*k*	V load bit matrix multiply register
0100m*m*	v*i*	000000	v*k*	11	101111	v*i* bmm(v*k*),m*m*	V bit matrix multiply
0100m*m*	v*i*	v*j*	v*k*	01	001100	v*i*{,d:,l:} cpys(v*j*,v*k*),m*m*	V copy sign 64-bit
0100m*m*	v*i*	v*j*	v*k*	00	001100	v*i*,{s:w} cpys(v*j*,v*k*),m*m*	V copy sign 32-bit
011100	000000	000000	000000	00	000100	vrip	Vector rest in peace

7.7.9 Vector Mask Instructions

g	i	j	k	t	f	Syntax	Description
0100m*m*	v*i*	v*j*	v*k*	01	101110	v*i*{,d:,l:} m*m*?v*k*:v*j*	V merge 64-bit
0101m*m*	v*i*	v*j*	s*k*	01	101110	v*i*{,d:,l:} m*m*?s*k*:v*j*	V/S merge 64-bit
0100m*m*	v*i*	v*j*	v*k*	00	101110	v*i*{,d:,l:} m*m*?v*k*:v*j*	V merge 32-bit
0101m*m*	v*i*	v*j*	s*k*	00	101110	v*i*,{s:w} m*m*?s*k*:v*j*	V/S merge 32-bit
010100	v*i*	000m*j*	a*k*	11	101100	v*i*{,l:} scan (a*k*,m*j*)	V iota continuous
010100	v*i*	000m*j*	a*k*	11	101101	v*i*{,l:} cidx(a*k*,m*j*)	V iota compressed
010000	v*i*	000m*j*	v*k*	01	010101	v*i*{,d:,l:} cmprss(v*k*,m*j*)	V compress
011100	000m*i*	000000	a*k*	00	001000	m*i* fill(a*k*)	VM fill
011100	a*i*	000000	000m*k*	00	001001	a*i* last(m*k*)	VM last element set
011100	a*i*	000000	000m*k*	00	001010	a*i* first(m*k*)	VM first element set
011100	a*i*	000000	000m*k*	00	001011	a*i* pop(m*k*)	VM pop count
011100	000m*i*	000m*j*	000m*k*	00	010000	m*i* m*j*&m*k*	VM logical AND
011100	00m*i*	00m*j*	00m*k*	00	010001	m*i* m*j*|m*k*	VM logical OR
011100	00m*i*	000m*j*	000m*k*	00	010010	m*i* ~m*j*^m*k*	VM logical equivalence

g	i	j	k	t	f	Syntax	Description
011100	00m*i*	00m*j*	00m*k*	00	010011	m*i* m*j*^m*k*	VM logical exclusive OR
011100	00m*i*	00m*j*	00m*k*	00	010100	m*i* m*j*&m*j*	VM logical ANDNOT

7.7.10 Vector Memory Instructions

The *hnt* (hint) fields in this section advises the hardware on whether and how a data value should be cached. The values have the following meanings:

ex (000) When you specify ex or do not specify a value for *hnt* you get the default, which is exclusive. Read misses allocate the cache line exclusively, anticipating a subsequent write by the same MSP. If the cache line is already shared, read misses allocate the line in shared mode. Write misses must allocate the line exclusively.

sh (001) Shared. Read misses allocate the cache line in the shared state, anticipating a subsequent read by a different MSP. This hint is ignored by writes.

na (010) Non-allocate. Read and write references do not allocate space for the line in local cache. This hint is used when no temporal locality is expected (to avoid cache pollution), or when the reference is to memory that will be subsequently accessed by a different MSP (explicit communication). If the referenced item is present in cache, the cache is accessed normally.

g	i	j	k	t	f	Syntax	Description
0110m*m*	v*i*	a*j*	a*k*	01	*hnt*000	v*i*{,d:,l:} [a*j*,a*k*],m*m*{,*hnt*}	V load 64-bit a*k* scaled by 8
0110m*m*	v*i*	a*j*	a*k*	00	*hnt*000	v*i*{,s:,w:} [a*j*,a*k*],m*m*{,*hnt*}	V load 32-bit a*k* scaled by 4
0110m*m*	v*i*	a*j*	imm6	01	*hnt*010	v*i*{,d:,l:} [a*j*,imm6],m*m*{,*hnt*}	V load 64-bit imm6 scaled by 8
0110m*m*	v*i*	a*j*	imm6	00	*hnt*010	v*i*{,s:,w:} [a*j*,imm6],m*m*{,*hnt*}	V load 32-bit imm6 scaled by 4
0110m*m*	v*i*	a*j*	v*k*	01	*hnt*100	v*i*{,d:,l:} [a*j*,v*k*],m*m*{,*hnt*}	V gather 64-bit v*k* scaled by 8
0110m*m*	v*i*	a*j*	v*k*	00	*hnt*100	v*i*{,s:,w:} [a*j*,v*k*],m*m*{,*hnt*}	V gather 32-bit v*k* scaled by 4

g	i	j	k	t	f	Syntax	Description
0110m*m*	v*i*	a*j*	a*k*	01	*hnt*001	[a*j*,a*k*] v*i*{,d:,l:},m*m*{,*hnt*}	V store 64-bit a*k* scaled by 8
0110m*m*	v*i*	a*j*	a*k*	00	*hnt*001	[a*j*,a*k*] v*i*{,s:,w:},m*m*{,*hnt*}	V store 32-bit a*k* scaled by 4
0110m*m*	v*i*	a*j*	imm6	01	*hnt*011	[a*j*,imm6] v*i*{,d:,l:},m*m*{,*hnt*}	V store 64-bit imm6 scaled by 8
0110m*m*	v*i*	a*j*	imm6	00	*hnt*011	[a*j*,imm6] v*i*{,s:,w:},m*m*{,*hnt*}	V store 32-bit imm6 scaled by 4
0110m*m*	v*i*	a*j*	v*k*	01	*hnt*101	[a*j*,v*k*] v*i*{,d:,l:},m*m*{,*hnt*}	V scatter 64-bit v*k* scaled by 8
0110m*m*	v*i*	a*j*	v*k*	00	*hnt*101	[a*j*,v*k*] v*i*{,s:,w:},m*m*{,*hnt*}	V scatter 32-bit v*k* scaled by 4
0110m*m*	v*i*	a*j*	v*k*	01	*hnt*111	[a*j*,v*k*] v*i*{,d:,l:},m*m*,ord{,*hnt*}	V ordered scatter 64-bit v*k* scaled by 8
0110m*m*	v*i*	a*j*	v*k*	00	*hnt*111	[a*j*,v*k*] v*i*{,s:,w:},m*m*,ord{,*hnt*}	V ordered scatter 32-bit v*k* scaled by 4

7.7.11 Privileged Instructions

g	i	j	k	t	f	Syntax	Description
000110	000000	000000	000000	00	000000	eret	Return from exception
000011	a*i*	a*j*	000000	01	000100	a*i* hw[a*i*]	Read hardware state
00011	000000	a*j*	a*k*	01	000101	hw[a*i*] a*k*	Write hardware state
011100	000000	000000	000000	00	000101	vclean	Clean hidden vector state

CAL Pseudo Instruction Descriptions [8]

This chapter lists the pseudo instructions presented throughout Chapter 3, page 33, in alphabetical order for easy reference.

Note: You can specify pseudo instructions in uppercase or lowercase, but not in mixed case.

Throughout this chapter, pseudo instructions with ignored fields (location or operand) are defined as follows:

```
ignored          pseudox
```

ignored — The assembler ignores the label field of this statement. If the field is not empty, then all of the characters in the field are skipped until a blank character is encountered and a caution-level message is issued. The first nonblank character following the blank character is assumed to be the beginning of the result field.

pseudox — Name of the Pseudo instruction with a blank label field.

```
          pseudoy        ignored
```

pseudoy — Name of the Pseudo instruction with a blank operand field.

ignored — The assembler ignores the operand field of this statement. If the field is not empty, then all of the characters in the field are skipped until a blank character is encountered and a caution-level message is issued. The first nonblank character following the blank character is assumed to be the beginning of the comment field.

8.1 Equate Symbol (=)

The equate symbol (=) when used as a pseudo instruction defines a symbol with the value and attributes determined by the expression. The symbol is not redefinable.

You can specify the = pseudo instruction anywhere within a program segment. If the = pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the = pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the = pseudo instruction is as follows:

```
[symbol]    =    expression[, [attribute]]
```

The *symbol* variable represents an optional unqualified symbol. The symbol is implicitly qualified by the current qualifier. The symbol must not be defined already. The label field can be blank. *symbol* must satisfy the requirements for symbols as described in Section 6.2, page 70.

All symbols found within *expression* must have been previously defined. The *expression* operand must meet the requirements for an expression as described in Section 6.9, page 94.

The *attribute* variable specifies a byte, word (W), or value (V) attribute. If present, it is used instead of the expression's attribute. If a byte-address attribute is specified, an expression with a word-address attribute is multiplied by 8; if a word-address attribute is specified, an expression with byte-address attribute is divided by 8. You cannot specify a relocatable expression as having value attribute.

In the following example, the symbol SYMB is assigned the value of (A*B+100)/4. The following illustrates the use of the = pseudo instruction:

```
      SYMB  =          (A*B+100)/4
```

8.2 (Deferred implementation) ALIGN

8.3 BASE

The BASE pseudo instruction specifies the base of numeric data as octal, decimal, or mixed when the base is not explicitly specified by an O', D', or X' prefix. The default is decimal.

You can specify the BASE pseudo instruction anywhere in a program segment. However, if the BASE pseudo instruction is located within a definition or skipping section, it is not recognized as a pseudo instruction.

The format of the BASE pseudo instruction is as follows:

```
ignored      BASE      option/*
```

The *option* variable specifies the numeric base of numeric data. It is a required single character specified as follows:

- O or o (Octal)
- D or d (Decimal)
- X or x (Hex)
- M or m (Mixed)

Numeric data is assumed to be octal, except for numeric data used for the following (assumed to be decimal):

- Statement counts in DUP and conditional statements
- Line count in the SPACE pseudo instruction
- Bit position or count in the BITW, BITB, or VWD pseudo instructions
- Character counts as in CMICRO, MICRO, OCTMIC, and DECMIC pseudo instructions
- Character count in data items (see Section 6.5.3, page 84).

When the asterisk (*) is used with the BASE pseudo instruction, the numeric base reverts to the base that was in effect prior to the specification of the current prefix within the current program segment. Each occurrence of a BASE pseudo instruction other than BASE * can modify the current prefix. Each BASE * releases the most current prefix and reactivates the prefix that preceded the current prefix. If all BASE pseudo instructions specified are released, a caution-level message is issued, and the default mode (decimal) is used.

The following example illustrates the use of the BASE pseudo instruction:

```
BASE    0 ; Change base from default to octal.
VWD     50/12       ; Field size and constant value both octal.
.
.
.
BASE    D           ; Change base from octal to decimal.
VWD     49/19       ; Field size and constant value both decimal.
.
.
.
BASE    M           ; Change from decimal to mixed base.
VWD     39/12       ; Field size decimal, constant value octal.
```

```
.
.
.
BASE    *          ; Resume decimal base.
BASE    *          ; Resume octal base.
BASE    *          ; Stack empty - resume decimal base (default)
```

8.4 BITW

The BITW pseudo instruction resets the current bit position to the value specified, relative to bit 0 of the current word. If the current bit position is not bit 0, a value of 64 (decimal) forces the following instruction to be assembled at the beginning of the next word (force word boundary). If the current bit position is bit 0, the BITW pseudo instruction with a value of 64 does not force a word boundary, and the instruction following BITW is assembled at bit 0 of the current word.

If the origin and location counters are set lower than the current value, any code previously generated in the overlapping part of the word is ORed with any new code.

The BITW pseudo instruction is restricted to sections that allow data or instructions and data. If the BITW pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BITW pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the BITW pseudo instruction is as follows:

```
ignored     BITW     [expression]
```

The *expression* variable is optional. If *expression* is not specified, the default is the absolute value of 0. If *expression* is specified, it must have an address attribute of value, a relative attribute of absolute, and be a positive value in the range from 0 to 64 (decimal). All symbols within *expression* (if any) must have been defined previously. If the current base is mixed, decimal is used.

The *expression* operand must meet the requirements for expressions as described in Section 6.9, page 94.

The value generated in the code field of the listing is equal to the value of the expression.

The following example illustrates the use of the BITW pseudo instruction:

```
BITW        D'39
```

8.5 BSS

The BSS pseudo instruction reserves a block of memory in a section. A forced byte boundary occurs, and the number of bytes specified by the operand field expression is reserved. This pseudo instruction does not generate data. To reserve the block of memory, the location and origin counters are increased.

You must specify the BSS pseudo instruction from within a program module. If the BSS pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BSS pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the BSS pseudo instruction is as follows:

```
[symbol]    BSS    [expression]
```

The *symbol* variable is optional. It is assigned the byte address of the location counter after the force byte boundary occurs. *symbol* must meet the requirement for symbols as described in Section 6.2, page 70.

The *expression* variable is an optional absolute expression with a byte address or value attribute and with all symbols, if any, previously defined. The value of the expression must be positive. A force byte boundary occurs before the expression is evaluated.

The *expression* operand must meet the requirements for an expression as described in Section 6.9, page 94.

The left margin of the listing shows the hex byte count.

The following example illustrates the use of the BSS pseudo instruction:

```
    BSS     4
A   CON     'NAME'
    CON     1
    CON     2
    BSS     16+A-W.*  ; Reserve more words so that the total
                      ; starting at A is 16.
```

8.6 BSSZ

The BSSZ pseudo instruction generates a block of bytes that contain 0's. When BSSZ is specified, a forced byte boundary occurs, and the number of zeroed bytes specified by the operand field expression is generated.

The BSSZ pseudo instruction is restricted to sections that have a type of data or instructions and data. If the BSSZ pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BSSZ pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the BSSZ pseudo instruction is as follows:

[*symbol*]	BSSZ	[*expression*]

The *symbol* variable represents an optional symbol. It is assigned the byte-address value of the location counter after the force byte boundary occurs. *symbol* must meet the requirements for a symbol as described in Section 6.2, page 70.

The *expression* variable represents an optional absolute expression with an attribute of byte address or value and with all symbols previously defined. The expression value must be positive and specifies the number of bytes containing 0's that will be generated. A blank operand field results in no data generation. The *expression* operand must meet the requirement for an expression as described in Section 6.9, page 94.

The octal word count of a BSSZ is shown in the left margin of the listing.

8.7 CMICRO

The CMICRO pseudo instruction assigns a name to a character string. After the name is defined, it cannot be redefined. If the CMICRO pseudo instruction is defined within the global definitions part of a program segment, it can be referenced at any time after its definition by any of the segments that follow. If the CMICRO pseudo instruction is defined within a program module, it can be referenced at any time after its definition within the module. However, a constant micro defined within a program module is discarded at the end of the module and cannot be referenced by any segments that follow.

If the CMICRO pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the CMICRO pseudo instruction

is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the CMICRO pseudo instruction is as follows:

name	CMICRO	[*string*[, [*exp*][, [*exp*], [*case*]]]]

The *name* variable is required and is assigned to the character string in the operand field. It has attributes that cannot be redefined. If *name* was previously defined and the string represented by the previous definition is not the same string, an error message is issued and definition occurs. If the strings match, no error message is issued and no definition occurs. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

The *string* variable represents an optional character string that can include previously defined micros. If *string* is not specified, an empty string is used. A character string can be delimited by any character other than a space. Two consecutive occurrences of the delimiting character indicate a single such character (for example, a micro consisting of the single character * can be specified as '*' or ****).

The *exp* variable represents optional expressions. The first expression must be an absolute expression that indicates the number of characters in the micro character string. All symbols, if any, must be previously defined. If the current base is mixed, decimal is used for the expression. The expressions must meet the requirements for expressions as described in Section 6.9, page 94.

The micro character string is terminated by the value of the first expression or the final apostrophe of the character string, whichever occurs first. If the first expression has a 0 or negative value, the string is considered empty. If the first expression is not specified, the full value of the character string is used. In this case, the string is terminated by the final apostrophe.

The second expression must be an absolute expression indicating the micro string's starting character. All symbols, if any, must be defined previously. If the current base is mixed, decimal is used for the expression.

The starting character of the micro string begins with the character that is equal to the value of the second expression, or with the first character in the character string if the second expression is null or has a value of 1 or less.

The optional *case* variable denotes the way uppercase and lowercase characters are interpreted when they are read from *string*. Character conversion is restricted

to the letter characters (A - Z and a - z) specified in *string*. You can specify *case* in uppercase, lowercase, or mixed case, and it must be one of the following values:

- MIXED or mixed

  string is interpreted as you entered it and no case conversion occurs. This is the default.

- UPPER or upper

  All lowercase alphabetic characters in *string* are converted to their uppercase equivalents.

- LOWER or lower

  All uppercase alphabetic characters in *string* are converted to their lowercase equivalents.

8.8 COMMENT

The COMMENT pseudo instruction defines a character string of up to 256 characters that will be entered as an informational comment in the generated binary load module.

If the operand field is empty, the comment field is cleared and no comment is generated. If a comment is specified more than once, the most recent one is used. If the last comment differs from the previous comment, a caution-level message is issued.

If a subprogram contains more than one COMMENT pseudo instruction, the character string from the last COMMENT pseudo instruction goes into the binary load module.

You must specify the COMMENT pseudo instruction from within a program module. If the COMMENT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the COMMENT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the COMMENT pseudo instruction is as follows:

```
ignored    COMMENT    [del-char[string-of-ASCII]del-char]
```

The *del-char* variable designates the delimiter character. It must be a single matching character on both ends of the ASCII character string. A character string can be delimited by a character other than an apostrophe. Any ASCII character other than a space can be used. Two consecutive occurrences of the delimiting character indicate that a single such character will be included in the character string.

The *string-of ASCII* variable is an optional ASCII character string of any length.

The following example illustrates the use of the COMMENT pseudo instruction:

```
IDENT    CAL
COMMENT  'COPYRIGHT CRAY INC.  2002'
COMMENT  -CRAY X1 computer system-
COMMENT  @ABCDEF@@FEDCBA@
END
```

8.9 CON

The CON pseudo instruction generates one or more full 64-bit words of binary data. This pseudo instruction always causes a forced 64-bit word boundary.

The CON pseudo instruction is restricted to sections that have a type of data or instructions and data. If the CON pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the CON pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the CON pseudo instruction is as follows:

```
[symbol]    CON    [expression]{ , [expression]}
```

The *symbol* variable is an optional symbol. It is assigned the byte address value of the location counter after the force 64-bit word boundary occurs. *symbol* must meet the requirements for a symbol as described in Section 6.2, page 70.

The *expression* variable is an expression whose value will be inserted into one 64-bit word. If an expression is null, a single zero word is generated. A force word boundary occurs before any operand field expressions are evaluated. A double-precision, floating-point constant is not allowed. *expression* must meet the requirements for an expression as described in Section 6.9, page 94.

The following example illustrates the use of the CON pseudo instruction:

```
A     CON     O'7777017
      CON     A             ; Generates the
                            ; address of A.
```

8.10 DATA

The DATA pseudo instruction generates zero or more bits of code for each data item parameter found in the operand field. If a label exists in the label field, a forced word boundary occurs and the symbol is assigned an address attribute and the value of the current location counter.

If a label is not included in the label field, a forced word boundary does not occur.

The DATA pseudo instruction is restricted to sections that have a type of data, constants, or instructions and data. If the DATA pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DATA pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The length of the field generated for each data item depends on the type of constant involved. Data items produce zero or more bits of absolute value binary code, as follows:

Data item	Description
Floating	One or two binary words, depending on whether the data item is a single- or double-precision data item
Integer	One binary word
Character	Zero or more bits of binary code depending on the following: • Character set specified • Number of characters in the string • Character count (optional) • Character suffix (optional)

A word boundary is not forced between data items.

The format of the DATA pseudo instruction is as follows:

[*symbol*]	DATA	[*data_item*][, [*data_item*]]

The symbol variable represents an optional symbol that is assigned the word address of the location counter after a force byte boundary. If no symbol is present, a force word boundary does not occur. *symbol* must meet the requirements for a symbol as described in Section 6.2, page 70.

The *data_item* variable represents numeric or character data. *data_item* must meet the requirements for a data item as described in Section 6.5, page 82.

The DATA pseudo instruction works with the actual number of bits given in the data item.

In the following example, unlabeled data items are stored in the next available bit position (see Figure 9):

```
IDENT    EXDAT
DATA    'abcd'*      ; Unlabeled data item 1.
DATA    'efgh'       ; Unlabeled data item 2.
END
```

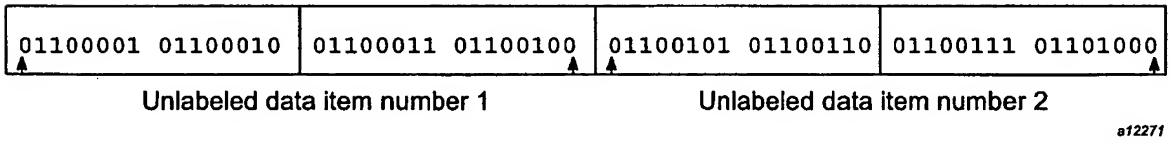

Figure 9. Storage of Unlabeled Data Items

In the following example, labeled data items cause a forced word boundary (see Figure 10):

```
        IDENT  EXDAT
        DATA   'abcd'*  ; Unlabeled data item 1.
ALPHA   DATA   'efgh'*  ; Labeled data item 1.
BETA    DATA   'ijkl'*  ; Labeled data item 2.
        DATA   'mnop'   ; Unlabeled data item 2.
```

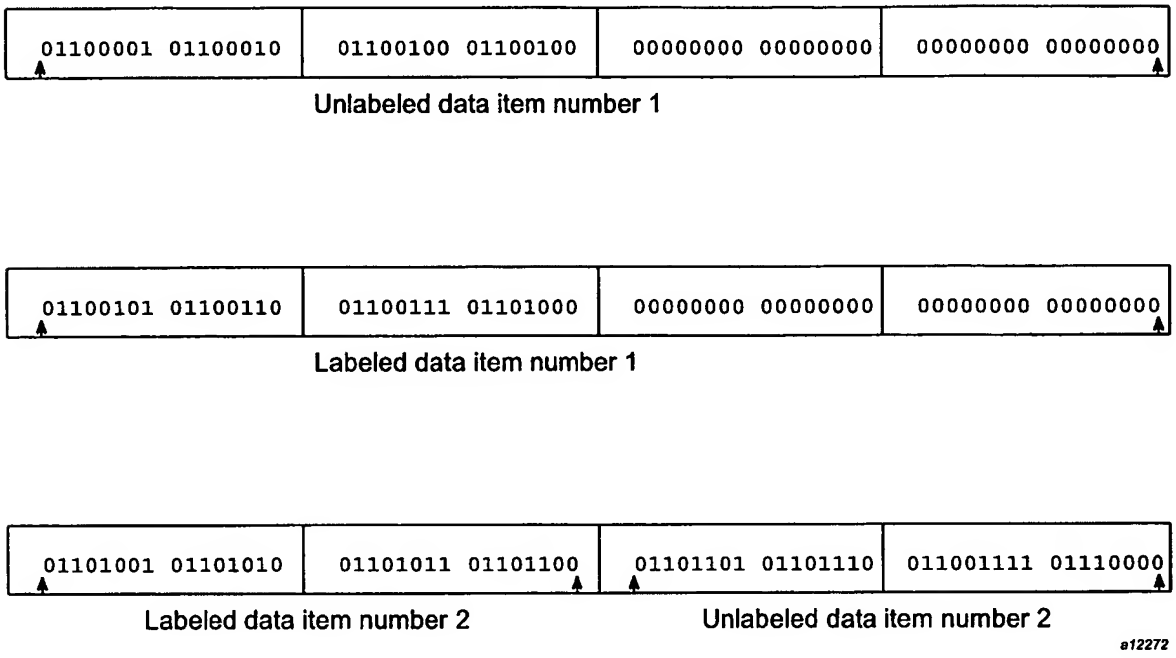

Figure 10. Storage of Labeled and Unlabeled Data Items

In the following example, if no forced word boundary occurs, data is stored bit by bit in consecutive words.

The following example shows the code generated by each source statement:

```
        IDENT  EXAMPLE
        DATA   0'5252,A'ABC'R  ; 00000000000000005252
                               ; 00000000000020241103
        DATA   'ABCD'          ; 0405022064204010020040
        DATA   'EFGH'          ; 0425062164404010020040
        DATA   'ABCD'*         ;  040502206420
        DATA   'EFGH'*         ;  10521443510
        DATA   'ABCD'12R       ; 00000000000000000000
                               ;  040502206420
        DATA   'EFGHIJ'*       ;            10521443510
                               ;  044512
LL2     DATA   'ABCD'          ; 0405022064204010020040
```

```
        DATA    100             ; 0000000000000000000144
        DATA    1.25E-9         ; 0377435274616704302142
        DATA    'THIS IS A MESSAGE'*L
                                ; 0521102225144022251440
                                ; 0404402324252324640507
                                ;  0424
        VWD     8/0             ;  000
        END
```

8.11 DBSM

The DBSM pseudo instruction generates a named label entry in the debug symbol tables with the type specified.

The format of the DBSM pseudo instruction is as follows:

```
[ignored]      DBSM      TYPE=symbol
```

TYPE is specified as either ATP or BOE (after the prologue or beginning of epilogue). *symbol* is user defined and marks these two points in the code. The *symbol* can appear anywhere in the code, but the address that is entered into the debug symbol table is the address of where the pseudo instruction appears in the code. This pseudo instruction is ignored unless you specify the debug option on the command line.

The following example illustrates the use of the DBSM pseudo instruction:

```
        IDENT   TEST
        ENTRY   FRED
FRED    =       *
        BSSZ    16          ; Fake prolog.
        S4      S4
CHK     =       *
        DBSM    ATP=FRED    ; Should be the same as CHK address.
        A1      S1
        A1      S1
        A1      S1
        DBSM    BOE=FRED    ; Address should be the same as the next
                            ; instruction
        S1      5
        J       B00
```

From the debugger, you can do a stop in FRED to generate a breakpoint at CHK. A call to this routine from a program executing in the debugger stops the execution.

8.12 DECMIC

The DECMIC pseudo instruction converts the positive or negative value of an expression into a positive or negative decimal character string that is assigned a redefinable micro name. The final length of the micro string is inserted into the code field of the listing.

You can specify DECMIC with zero, one, or two expressions. DECMIC converts the value of the first expression into a character string with a character length indicated by the second expression. If the second expression is not specified, the minimum number of characters needed to represent the decimal value of the first expression is used.

If the second expression is specified, the string is equal to the length specified by the second expression. If the number of characters in the micro string is less than the value of the second expression, and the value of the first expression is positive, the character value is right-justified with the specified fill characters (zeros or blanks) preceding the value.

If the number of characters in the string is less than the value of the second expression, and the value of the first expression is negative, a minus sign precedes the value. If zero fill is indicated, zeros are used as fill between the minus sign and the value. If blank fill is indicated, blanks are used as fill before the minus sign.

If the number of characters in the string is greater than the value of the second expression, the characters at the beginning of the string are truncated and a warning message is issued.

You can specify the DECMIC pseudo instruction anywhere within a program segment. If the DECMIC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DECMIC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the DECMIC pseudo instruction is as follows:

name DECMIC [[[*expression*$_1$] , *expression*$_2$[, [*option*]]]]

name is assigned to the character string that represents the decimal value of *expression*$_1$ and has redefinable attributes. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

expression1 is optional and represents the micro string equal to the value of the expression. If specified, *expression*$_1$ must have an address attribute of value and a relative attribute of absolute with all symbols, if any, previously defined. If the first expression is not specified, the absolute value of 0 is used. If the current base is mixed, a default of octal is used. If the first expression is not specified, the absolute value of 0 is used when creating the micro string. The *expression*$_1$ operand must meet the requirements for expressions as described in Section 6.9, page 94.

expression2 is optional and provides a positive character count less than or equal to decimal 20. If this parameter is present, the necessary leading zeros or blanks (depending on *option*) are supplied to provide the requested number of characters. If specified, *expression*$_2$ must have an address attribute of value and a relative attribute of absolute with all symbols, if any, previously defined. If the current base is mixed, a default of decimal is used. *expression*$_2$ must meet the requirements for expressions as described in Section 6.9, page 94.

If *expression*$_2$ is not specified, the micro string is represented in the minimum number of characters needed to represent the decimal value of the first expression.

option represents the type of fill characters (ZERO for zeros or BLANK for spaces) to be used if the second expression is present and fill is needed. The default is ZERO. You can enter *option* in mixed case.

The following example illustrates the use of the DECMIC and MICSIZE pseudo instructions:

```
MIC  MICRO  'ABCD'
V  MICSIZE  MIC          ; The value of V is the number of
                         ; characters in the micro string
                         ; represented by MIC.
DECT  DECMIC  V,2        ; DECT is a micro name.
_*There are "DECT" characters in MIC.
* There are 19 characters in MIC.[1]
```

1 Generated by the assembler

The following example demonstrates the ZERO and BLANK options with positive and negative strings:

```
      BASE  D       ; The base is decimal
ONE   DECMIC 1,2
_*    "ONE"                       ; Returns 1 in 2 digits.
*     01                          ; Returns 1 in 2 digits.
TWO   DECMIC 5*8+60+900,3         ; Decimal 1000.
_*    "TWO"                       ; Returns 1000 as 3 digits (000).
*     000                         ; Returns 1000 as 3 digits (000).
THREE DECMIC -256000,10,ZERO      ; Decimal string with zero fill.
_*    "THREE"                     ; Minus sign, zero fill, value.
*     -000256000                  ; Minus sign, zero fill, value.
FOUR  DECMIC -256000,10,BLANK     ; Decimal string with blank fill.
_*    "FOUR"                      ; Blank fill, minus sign, value.
*     ^^^-256000                  ; Blank fill, minus sign, value.
FIVE  DECMIC 256000,10,ZERO
_*    "FIVE"                      ; Zero fill on the left.
*     0000256000                  ; Zero fill on the left.
SIX   DECMIC 256000,10,BLANK
_*    "SIX"                       ; Blank fill (^) on the left.
*     ^^^^256000                  ; Blank fill (^) on the left.
      END
SEVEN DECMIC 256000,5
_*    "SEVEN"                     ; Truncation warning issued.
*     56000                       ; Truncation warning issued.
EIGHT DECMIC 777777777,3
_*    "EIGHT"                     ; Truncation warning issued.
*     777                         ; Truncation warning issued.
```

8.13 DMSG

The DMSG pseudo instruction issues a comment level diagnostic message that contains the string found in the operand field, if a string exists. If the string consists of more than 80 characters, a warning message is issued and the string is truncated.

Comment level diagnostic messages might not be issued by default on the operating system in which the assembler is executing. For more information, see Chapter 5, page 47.

The assembler recognizes up to 80 characters within the string, but the string may be truncated further when the diagnostic message is issued (depending on the operating system in which the assembler is executing).

You can specify the DMSG pseudo instruction anywhere within a program segment. If the DMSG pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DMSG pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the DMSG pseudo instruction is as follows:

```
ignored     DMSG     [del-char[string-of-ASCII]del-char]
```

The *del-char* variable represents the delimiting character. It must be a single matching character on both ends of the ASCII character string. Apostrophes and spaces are not legal delimiters; all other ASCII characters are allowed. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The *string-of ASCII* variable represents the ASCII character string that will be printed to the diagnostic file. A maximum of 80 characters is allowed.

> Note: Using the DMSG pseudo instruction for assembly timings can be deceiving. For example, if the DMSG pseudo instruction is inserted near the beginning of an assembler segment, more time could elapse (from the time that the assembler begins assembling the segment to the time the message is issued) than you might have expected.

8.14 DUP

The DUP pseudo instruction introduces a sequence of code that is assembled repetitively a specified number of times The duplicated code immediately follows the DUP pseudo instruction.

The DUP pseudo instruction is described in detail in Section 9.4, page 248.

8.15 ECHO

The ECHO pseudo instruction defines a sequence of code that is assembled zero or more times immediately following the definition.

The ECHO pseudo instruction is described in detail in Section 9.5, page 250.

8.16 EDIT

The EDIT pseudo instruction toggles the editing function on and off within a program segment. Appending (\ in the new format) and continuation (, in the old format) are not affected by the EDIT pseudo instruction. The current editing status is reset at the beginning of each segment to the editing option specified on the assembler invocation statement. For a description of statement editing, see Section 6.10, page 97.

You can specify the EDIT pseudo instruction anywhere within a program segment. If the EDIT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the EDIT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the EDIT pseudo instruction is as follows:

```
ignored     EDIT     */option
```

The *option* variable turns editing on and off. *option* can be specified in uppercase, lowercase, or mixed case, and it can be one of the following:

- ON (enable editing)
- OFF (disable editing)
- No entry (reverts to the format specified on the assembler invocation statement)

An asterisk (*) resumes use of the edit option in effect before the most recent edit option within the current program segment. Each occurrence of an EDIT other than an EDIT * initiates a new edit option. Each EDIT * removes the current edit option and reactivates the edit option that preceded the current edit option. If the EDIT * statement is encountered and all specified edit options were released, a caution-level message is issued and the default is used.

8.17 EJECT

The EJECT pseudo instruction causes the beginning of a new page in the output listing. EJECT is a list control pseudo instruction and by default, is not listed. To

include the EJECT pseudo instruction on the listing, specify the LIS option on the LIST pseudo instruction.

You can specify the EJECT pseudo instruction anywhere within a program segment. If the EJECT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the EJECT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the EJECT pseudo instruction is as follows:

```
ignored     EJECT     ignored
```

8.18 ELSE

The ELSE pseudo instruction terminates skipping initiated by the IFA, IFC, IFE, ELSE, or SKIP pseudo instructions with the same label field name. If statements are currently being skipped under control of a statement count, ELSE has no effect.

You can specify the ELSE pseudo instruction anywhere within a program segment. If the assembler is not currently skipping statements, ELSE initiates skipping. Skipping is terminated by an ELSE pseudo instruction with a matching label field name. If the ELSE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction.

The format of the ELSE pseudo instruction is as follows:

```
name     ELSE     ignored
```

The *name* variable specifies a required name for a conditional sequence of code. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

The following example illustrates the use of the ELSE pseudo instruction:

```
SYM     =       1
L       MICRO   'LESS THAN'
DEF     =       1000
BUF     =       100
        IFA     #DEF,A,1
A       =       10
```

```
BTEST    IFA          EXT,SYM
WARNING  ERROR            ; Generate warning message if SYM is
                          ; absolute.
BTEST    ELSE
         A1           SYM ; Assemble if SYM is not absolute.
BTEST    ENDIF
*Assemble BSSZ instruction if W.* is less than BUF, otherwise
*assemble ORG

         IFE          W.*,LT,BUF,2
         BSSZ         BUF-W.*
                          ; Generate words of zero to address BUF.
         SKIP         1    ; Skip next statement.
         ORG          BUF
         IFC          ' "L" ',EQ,,2
ERROR    ERROR             ; Error message if micro string defined
                          ; by L is empty.
X        IFC          'ABCD',GT,'ABC'
                          ; If ABCD is greater than ABC,
         S1           DEF  ; Statement is included.
         S2           BUF  ; Statement is included.
X        ENDIF
Y        IFC          ' ',GT,,2
                          ; If single space is greater than null
                          ; string,
         S3           DEF  ; Statement is included.
         S4           BUF  ; Statement is included.
Z        IFC          '''',EQ,*'*,2
                          ; If single apostrophe equals single
                          ; apostrophe.
         S5           5    ; Statement is included.
         S6           6    ; Statement is included.
Z        ENDIF
```

8.19 END

The END pseudo instruction terminates a program segment (module initiated with an IDENT pseudo instruction) under the following conditions:

- If the assembler is not in definition mode
- If the assembler is not in skipping mode

- If the END pseudo instruction does not occur within an expansion

The format of the END pseudo instruction is as follows:

```
ignored     END     ignored
```

If the END pseudo instruction is found within a definition, a skip sequence, or an expansion, a message is issued indicating that the pseudo instruction is not allowed within these modes and the statement is treated as follows:

- Defined if in definition mode
- Skipped if in skipping mode
- Do-nothing instruction if in an expansion

You can specify the END pseudo instruction only from within a program module. If the END pseudo instruction is valid and terminates a program module, it causes the assembler to take the following actions:

- Generates a cross-reference for symbols if the cross-reference list option is enabled and the listing is enabled
- Clears and resets the format option
- Clears and resets the edit option
- Clears and resets the message level
- Clears and resets all list control options
- Clears and resets the default numeric base
- Discards all qualified, redefinable, nonglobal, and %% symbols
- Discards all qualifiers
- Discards all redefinable and nonglobal micros
- Discards all local macros, opdefs, and local pseudo instructions (defined with an OPSYN pseudo instruction)
- Discards all sections

8.20 ENDDUP

The ENDDUP pseudo instruction ends the definition of the code sequence to be repeated. An ENDDUP pseudo instruction terminates a dup or echo definition with the same name.

The ENDDUP pseudo instruction is described in detail in Section 9.8, page 254.

8.21 ENDIF

The ENDIF pseudo instruction terminates skipping initiated by an IFA, IFE, IFC, ELSE, or SKIP pseudo instruction with the same label field name; otherwise, ENDIF acts as a do-nothing pseudo instruction. ENDIF does not affect skipping, which is controlled by a statement count.

You can specify the ENDIF pseudo instruction anywhere within a program segment. Skipping is terminated by an ENDIF pseudo instruction with a matching label field name. If the ENDIF pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction.

The format of the ENDIF pseudo instruction is as follows:

```
name    ENDIF    ignored
```

The *name* variable specifies a required name for a conditional sequence of code. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

Note: If an END pseudo instruction is encountered in a skipping sequence, an error message is issued and skipping is continued. You should not use the END pseudo instruction within a skipping sequence.

8.22 ENDM

An ENDM pseudo instruction terminates the body of a macro or opdef definition.

The ENDM pseudo instruction is described in detail in Section 9.6, page 252.

8.23 ENDTEXT

The ENDTEXT pseudo instruction terminates text source initiated by a TEXT instruction. An IDENT or END pseudo instruction also terminates text source.

The ENDTEXT is a list control pseudo instruction and by default, is not listed unless the TXT option is enabled. If the LIS option is enabled, the ENDTEXT instruction is listed regardless of other listing options.

You can specify the ENDTEXT pseudo instruction anywhere within a program segment. If the ENDTEXT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ENDTEXT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ENDTEXT pseudo instruction is as follows:

```
ignored     ENDTEXT     ignored
```

The following example illustrates the use of the ENDTEXT pseudo instruction (with the TXT option off).

The following represents the source listing:

```
      IDENT       TEXT
A     =               2
TXTNAME  TEXT       'An example.'
B     =               3
C     =               4
      ENDTEXT
      A1              A
      A2              B
      END
```

The following represents the output listing:

```
     IDENT  TEXT
A    =        2
TXTNAME TEXT  'An example.'
      A1       A
      A2       B
      END
```

8.24 ENTRY

The ENTRY pseudo instruction specifies symbolic addresses or values that can be referred to by other program modules linked by the loader. Each entry symbol must be an absolute, immobile, or relocatable symbol defined within the program module.

The ENTRY pseudo instruction is restricted to sections that allow instructions or data or both. If the ENTRY pseudo instruction is found within a definition or skipping sequence, it is defined and not recognized as a pseudo instruction.

The format of the ENTRY pseudo instruction is as follows:

```
ignored     ENTRY     [symbol:[attr]],[symbol:[attr]]
```

The *symbol* variable specifies the name of zero, one, or more symbols. Each of the names must be defined as an unqualified symbol within the same program module. The corresponding symbol must not be redefinable, external, or relocatable relative to either a stack or a task common section.

The length of the symbol is restricted depending on the type of loader table that the assembler is currently generating. If the symbol is too long, the assembler will issue an error message.

The *symbol* operand must meet the requirements for symbols as described in Section 6.2, page 70.

The *attr* attribute can be either OBJ or FUNC, specifying whether the symbol is an object or a function entry point. If *attr* is not specified and the ENTRY pseudo instruction occurs in a DATA, CONST, or COMMON section, the default is OBJ.

The following example illustrates the use of the ENTRY pseudo instruction:

```
        ENTRY   EPTNME,TREG
        .
        .
        .
EPTNME  =       *
TREG    =       O'17
```

8.25 ERRIF

The ERRIF pseudo instruction conditionally issues a listing message. If the condition is satisfied (true), the appropriate user-defined message is issued. If

the level is not specified, the ERRIF pseudo instruction issues an error-level message. If the condition is not satisfied (false), no message is issued. If any errors are encountered while evaluating the operand field, the resulting condition is handled as if true and the appropriate user-defined message is issued.

You can specify the ERRIF pseudo instruction anywhere within a program segment. If the ERRIF pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ERRIF pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ERRIF pseudo instruction is as follows:

[*option*] ERRIF [*expression*], *condition*, [*expression*]

The *option* variable used in the ERRIF pseudo instruction is the same as in the ERROR pseudo instruction. See the ERROR pseudo instruction for information.

Zero, one, or two expressions to be compared by *condition*. If one or both of the expressions are missing, a value of absolute 0 is substituted for every expression that is not specified. Symbols found in either of the expressions can be defined later in a segment.

The *expression* operand must meet the requirements for expressions as described in Section 6.9, page 94.

The *condition* variable specifies the relationship between two expressions that causes the generation of an error. For LT, LE, GT, and GE, only the values of the expressions are examined. You can enter *condition* in uppercase, lowercase, or mixed case, and it can be one of the following:

- LT (less than)

  The value of the first expression must be less than the value of the second expression.

- LE (less than or equal)

  The value of the first expression must be less than or equal to the value of the second expression.

- GT (greater than)

  The value of the first expression must be greater than the value of the second expression.

- GE (greater than or equal)

  The value of the first expression must be greater than or equal to the value of the second expression.

- EQ (equal)

  The value of the first expression must be equal to the value of the second expression. Both expressions must be one of the following:

  - Absolute
  - Immobile relative to the same section
  - Relocatable in the program section or the same common section
  - External relative to the same external symbol.

    The word-address, byte-address, or value attributes must be the same.

- NE (not equal)

  The first expression must not equal the second expression. Both expressions cannot be absolute, or external relative to the same external symbol, or relocatable in the program section or the same common section. The word-address, byte-address, or value attributes are not the same.

The ERRIF pseudo instruction does not compare the address and relative attributes. A CAUTION level message is issued.

The following example illustrates the use of the ERRIF pseudo instruction:

```
P       ERRIF   ABC,LT,DEF
```

8.26 ERROR

The ERROR pseudo instruction unconditionally issues a listing message. If the level is not specified, the ERROR pseudo instruction issues an error level message. If the condition is not satisfied (FALSE), no message is issued.

You can specify the ERROR pseudo instruction anywhere within a program segment. If the ERROR pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ERROR pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ERROR pseudo instruction is as follows:

```
[option]    ERROR    ignored
```

The *option* variable specifies the error level. It can be entered in upper, lower, or mixed case. The following error levels are mapped directly into a user-defined message of the corresponding level:

```
COMMENT, NOTE, CAUTION, WARNING, or ERROR
```

The following levels are mapped into an error-level message:

```
C, D, E, F, I, L, N, O, P, R, S, T, U, V, or X
```

The following levels are mapped into warning-level messages:

```
W, W1, W2, W3, W4, W5, W6, W7, W8, W9, Y1, or Y2
```

Messages C through Y2 provide compatibility with Cray Assembly Language, version 1 (CAL1).

The assembler can produce five similar messages with differing levels (error, warning, caution, note, or comment). The ERROR pseudo instruction can be used to check for valid input and to assign an appropriate message.

In the following example, a user-defined error level message is specified:

```
ERROR   ERROR  ; ***ERROR*** Input is not valid
```

8.27 EXITM

The EXITM pseudo instruction immediately terminates the innermost nested macro or opdef expansion, if any, caused by either a macro or an opdef call.

The EXITM pseudo instruction is described in detail in Section 9.7, page 253.

8.28 EXT

The EXT pseudo instruction specifies linkage to symbols that are defined as entry symbols in other program modules. They can be referred to from within the program module, but must not be defined as unqualified symbols elsewhere within the program module. Symbols specified in the EXT instruction are defined as unqualified symbols that have relative attributes of external and specified address.

You can specify the EXT pseudo instruction anywhere within a program module. If the EXT pseudo instruction is found within a definition or skipping sequence, it is defined and not recognized as a pseudo instruction.

The format of the EXT pseudo instruction is as follows:

```
ignored     EXT     [symbol:[attribute]],[symbol:[attribute]]
```

The variables associated with the EXT pseudo instruction are described as follows:

- *symbol*

  The *symbol* variable specifies the name of zero, one, or more external symbols. Each of the names must be an unqualified symbol that has a relative attribute of external and the corresponding address attribute.

  The length of the symbol is restricted depending on the type of loader table that the assembler is currently generating. If the symbol is too long, the assembler will issue an error message.

  The *symbol* operand must meet the requirements for symbols as described in Section 6.2, page 70.

- *attribute*

  The *attribute* variable specifies the attribute *symbol-type* as follows:

  - The *symbol-type* will be assigned to the external symbol; it can be one of the following:

V or v	Value (default)
OBJ	Object
FUNC	Function

  - The *linkage-attribute* type is the linkage attribute that will be assigned to the external symbol. Linkage attributes can be specified in uppercase, lowercase, or mixed case, and they can be one of the following:

    ```
    HARD (default)
    SOFT
    ```

  If the *linkage-attribute* is not specified on the EXT pseudo instruction, the default is HARD. All hard external references are resolved at load time.

A soft reference for a particular external name is resolved at load time only when at least one other module has referenced that same external name as a hard reference.

You conditionally reference a soft external name at execution time. If a soft external name was not included at load time and is referenced at execution time, an appropriate message is issued.

If the operating system for which the assembler is generating code does not support soft externals, a caution-level message is issued and soft externals are treated as hard externals.

Note: Typically, a soft external is used for references to large software packages (such as graphics packages) that may not be required in a particular load. When such code is required, load time is shorter and the absolute module is smaller in size. For most uses, however, hard externals are recommended.

The following example illustrates the use of the EXT pseudo instruction:

```
        IDENT  A
        .
        .
        .
        ENTRY  VALUE
VALUE   =      2.0
        .
        .
        .
        END
        IDENT B
        EXT    VALUE
        CON    VALUE    ; The 64-bit external. External value 2.0 is
                        ; stored here by the loader.
        END
```

8.29 FORMAT

The assembler supports both the CAL version 1 (CAL1) statement format and a new statement format. The FORMAT pseudo instruction lets you switch between statement formats within a program segment. The current statement format is reset at the beginning of each section to the format option specified on the assembler invocation statement. For a description of the recommended formatting conventions for the new format, see Section 6.2.1.2, page 72.

You can specify the FORMAT pseudo instruction anywhere within a program segment. If the FORMAT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the FORMAT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the FORMAT pseudo instruction is as follows:

```
ignored     FORMAT     */option
```

The *option* variable specifies old or new format. *option* can be specified in uppercase, lowercase, or mixed case, and it can be one of the following:

- OLD (old format)
- NEW (new format)
- No entry (reverts to the EDIT option specified on the assembler invocation statement)

An asterisk (*) resumes use of the format option in effect before the most recent format option within the current program segment. Each occurrence of a FORMAT other than a FORMAT * initiates a new format option. Each FORMAT * removes the current format option and reactivates the format that preceded the current format. If the FORMAT * statement is encountered and all specified format options were released, a caution-level message is issued and the default is used.

8.30 IDENT

The IDENT pseudo instruction identifies a program module and marks its beginning. The module name appears in the heading of the listing produced by the assembler (if the title pseudo instruction has not been used) and in the generated binary load module.

You must specify the IDENT pseudo instruction in the global part of a CAL program. If the IDENT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IDENT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the IDENT pseudo instruction is as follows:

```
ignored     IDENT     lname
```

The *lname* variable is the long name of the program module. *lname* must meet the requirements for long names as described in Section 6.3, page 76.

The length of the long name is restricted depending on the type of loader table the assembler is currently generating. If the name is too long, the assembler issues an error message.

The following example illustrates the use of the IDENT pseudo instruction:

```
IDENT   EXAMPLE    ; Beginning of the EXAMPLE program module
.
.                  ; Other code goes here
.
END                ; End of the EXAMPLE program module
```

8.31 IFA

The IFA pseudo instruction tests an attribute of an expression. If the expression has the specified attribute, assembly continues with the next statement. If the result of the attribute test is false, subsequent statements are skipped. If a label field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered; otherwise, skipping stops when the statement count is exhausted.

If any errors are encountered while evaluating the attribute-condition, the resulting condition is handled as if true and the appropriate listing message is issued.

You can specify the IFA pseudo instruction anywhere within a program segment. If the IFA pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IFA pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the IFA pseudo instruction is as follows:

[*name*]	IFA	[# or !]*exp-attribute, expression*[, [*count*]]
[*name*]	IFA	[# or !]*redef-attribute,symbol*[, [*count*]]
[*name*]	IFA	[# or !]*reg-attribute,reg-arg_value*[,[*count*]]
[*name*]	IFA	[# or !]*micro-attribute, mname*[,[*count*]]

The *name* variable specifies an optional name of a conditional sequence of code. A conditional sequence of code that is controlled by a name is ended by an ENDIF pseudo instruction with a matching name. To reverse the condition of a conditional sequence of code controlled by a name, use an ELSE pseudo instruction with a matching name. If both *name* and *count* are present, name takes precedence. *name* must meet the requirements for names as described in Section 6.3, page 76

Either the pound sign (#) or the exclamation mark (!) is optional and negates the condition. If errors occur in the attribute condition, the condition is evaluated as if it were true. Although # or ! does not change the condition, it does specify the if not condition.

The *exp-attribute* variable is a mnemonic that signifies an attribute of *expression*. *expression* must meet the requirement for an expression as described in Section 6.9, page 94.

An expression has only one address attribute (VAL, PA, or WA) and relative attribute (ABS, IMM, REL, or EXT). An attribute also can be any of the following mnemonics preceded by a complement sign (#), indicating that the second subfield does not satisfy the corresponding condition. You can specify all of the following mnemonics in mixed case:

Mnemonic	Attribute
VAL	Value; requires all symbols within the expression to be defined previously.
PA	byte address; requires all symbols, if any, within the expression to be defined previously.
WA	Word address; requires all symbols, if any, within the expression to be defined previously.
ABS	Absolute; requires all symbols, if any, within the expression to be defined previously.
IMM	Immobile; requires all symbols, if any, within the expression to be defined previously.
REL	Relocatable; requires all symbols, if any, within the expression to be defined previously.
EXT	External; requires all symbols, if any, within the expression to be defined previously.
CODE	Immobile or relocatable; relative to a code section. CODE requires all symbols, if any, within the expression to be defined previously.

DATA	Immobile or relocatable; relative to a data section. DATA requires all symbols, if any, within the expression to be defined previously.
ZERODATA	Immobile or relocatable; relative to a zero data section. ZERODATA requires all symbols, if any, within the expression to be defined previously.
CONST	Immobile or relocatable; relative to a constant section. CONST requires all symbols, if any, within the expression to be defined previously.
MIXED	Immobile or relocatable; relative to a common section. MIXED requires all symbols, if any, within the expression to be defined previously.
COM	Immobile or relocatable; relative to a common section. COM requires all symbols, if any, within the expression to be defined previously.
COMMON	Immobile or relocatable; relative to a common section. COMMON requires all symbols, if any, within the expression to be defined previously.
TASKCOM	Immobile or relocatable; relative to a task common section. TASKCOM requires all symbols, if any, within the expression to be defined previously.
ZEROCOM	Immobile or relocatable; relative to a zero common section. ZEROCOM requires all symbols, if any, within the expression to be defined previously.
DYNAMIC	Immobile or relocatable; relative to a dynamic section. DYNAMIC requires all symbols, if any, within the expression to be defined previously.
STACK	Immobile or relocatable; relative to a stack section. STACK requires all symbols, if any, within the expression to be defined previously.
CM	Immobile or relocatable; relative to a section that is placed into common memory. CM requires all symbols, if any, within the expression to be defined previously.
EM	Immobile or relocatable; relative to a section that is placed into extended memory. EM requires all symbols, if any, within the expression to be defined previously. If EM is specified, the condition always fails.

LM	Immobile or relocatable; relative to a section that is placed into local memory. LM requires all symbols, if any, within the expression to be defined previously. If LM is specified for a Cray system, the condition always fails.
DEF	True if all symbols in the expression were defined previously; otherwise, the condition is false.

The *redef-attribute* variable specifies a redefinable attribute. The condition is true if the symbol following *redef-attribute* is redefinable; otherwise, the condition is false. Redefinable attribute is defined as follows:

Mnemonic	Attribute
SET	The *symbol* in the second subfield is a redefinable symbol. *symbol* must meet the requirements for a symbol as described in Section 6.2, page 70.

The *reg-attribute* variable specifies a register attribute. *reg-arg-value* is any ASCII character up to but not including a legal terminator (blank character or semicolon; new format) and element separator character (,). If you specify REG, the condition is true if the following string is a valid complex-register; otherwise, the condition is false. Register-attribute is defined as follows:

Mnemonic	Attribute
REG	The second subfield contains a valid A, S, or C register designator.

The *micro-attribute* variable specifies an attribute of the micro specified by *mname*. *mname* must meet the requirements for identifiers as described in Section 6.3, page 76. If you specify MIC, the condition is true if the following identifier is an existing micro name; otherwise, the condition is false. *micro-attribute* is defined as follows:

Mnemonic	Attribute
MIC	The name in the second subfield is a micro name.
MICRO	The name in the second subfield is a micro name and the corresponding micro can be redefined.
CMICRO	The name in the second subfield is a micro name and the corresponding micro is constant.

The *count* variable specifies the statement count. It must be an absolute expression with positive value. All symbols in the expression, if any, must be

previously defined. A missing or null count subfield gives a zero count. *count* is used only when the label field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

The following example illustrates the use of the IFA pseudo instruction:

```
SYM1   SET  1
SYM2   =    2
       IFA  SET,SYM1,2      ; If the condition is true,
       S1   SYM1            ; include this statement
       S2   SYM2            ; include this statement
SYM2   =    1
       IFA  SET,SYM2,1      ; If the condition is false,
       S3   SYM2            ; skip this statement.
```

8.32 IFC

The IFC pseudo instruction tests a pair of character strings for a condition under which code will be assembled if the relation specified by *condition* is satisfied (true). If the relationship is not satisfied (false), subsequent statements are skipped. If a label field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered; otherwise, skipping stops when the statement count is exhausted.

If any errors are encountered during evaluation of the string condition, the resulting condition is handled as if true and an appropriate listing message is issued.

You can specify the IFC pseudo instruction anywhere within a program segment. If the IFC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IFC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the IFC pseudo instruction is as follows:

[*name*] IFC [*string*], *condition*, [*string*] [, [*count*]]

The *name* variable specifies an optional name of a conditional sequence of code. A conditional sequence of code that is controlled by a name is ended by an ENDIF pseudo instruction with a matching name. To reverse the condition of a conditional sequence of code controlled by a name, use an ELSE pseudo instruction with a matching name. If both *name* and *count* are present, *name*

takes precedence. *name* must meet the requirements for names as described in Section 6.3, page 76.

The *string* variable specifies the character string that will be compared. The first and third subfields can be null (empty) indicating a null character string. The ASCII character code value of each character in the first string is compared with the value of each character in the second string. The comparison is from left to right and continues until an inequality is found or until the longer string is exhausted. A value of 0 is substituted for missing characters in the shorter string. Micros and formal parameters can be contained in the character strings.

The *string* operand is an optional ASCII character string that must be specified with one matching character on both ends. A character string can be delimited by any ASCII character other than a comma or space. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The following example compares the character strings `O'100` and `ABCD*`:

```
AIF IFC =O'100=,EQ,*ABCD***
```

The condition variable specifies the relation that will be satisfied by the two strings. You can enter *condition* in mixed case, and it must be one of the following:

- `LT` (less than)

  The value of the first string must be less than the value of the second string.

- `LE` (less than or equal)

  The value of the first string must be less than or equal to the value of the second string.

- `GT` (greater than)

  The value of the first string must be greater than the value of the second string.

- `GE` (greater than or equal)

  The value of the first string must be greater than or equal to the value of the second string.

- `EQ` (equal)

  The value of the first string must be equal to the value of the second string.

- `NE` (not equal)

The value of the first string must not equal the value of the second string.

The *count* variable specifies the statement count. It must be an absolute expression with positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count. The *count* operand is used only when the label field is not specified. If name is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

The following examples illustrates the use of the IFC pseudo instruction. The first string is delimited by the at sign (@), and the second string is delimited by the percent sign (%). The first string is equal to the second string.

```
     IDENT TEST
EX1  IFC   @ABC@@D@,EQ,%ABC@D%
                              ; The condition is true.
                              ; Skipping does not occur.
     S1    1                  ; Statement is included.
     S2    2                  ; Statement is included.
EX1  ELSE                     ; Statements within the ELSE sequence
                              ; are included only if the condition
                              ; fails.
     S3    3                  ; Statement is skipped.
EX1  ENDIF                    ; End of skip sequence.
     END
```

In the next example, the first string is not equal to the second string, the two statements following the IFC are skipped.

```
     IDENT TEST
EX1  IFC   @ABBCD@,EQ,@ABCD@  ; The condition is false.
                              ; Skipping occurs.
     S1    1                  ; Statement is skipped.
     S2    2                  ; Statement is skipped.
EX1  ENDIF                    ; End of skip sequence
     S3    3                  ; This statement is included regardless
                              ; of whether the condition is true or
                              ; false.
     END
```

8.33 IFE

The IFE pseudo instruction tests a pair of expressions for a condition. If the relation *(condition)* specified by the operation is satisfied, code is assembled. If

condition is true, assembly resumes with the next statement; if *condition* is false, subsequent statements are skipped. If a label field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered; otherwise, skipping stops when the statement count is exhausted.

If any errors are encountered during the evaluation of the expression-condition, the resulting condition is handled as if true and an appropriate listing message is issued.

If an assembly error is detected, assembly continues with the next statement.

You can specify the `IFE` pseudo instruction anywhere within a program segment. If the `IFE` pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the `IFE` pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the `IFE` pseudo instruction is as follows:

```
[name]      IFE      [expression], condition, [expression]  [, [count]]
```

The *name* variable specifies an optional name of a conditional sequence of code. A conditional sequence of code that is controlled by a name is ended by an `ENDIF` pseudo instruction with a matching name. To reverse the condition of a conditional sequence of code controlled by a name, use an `ELSE` pseudo instruction with a matching name. If both *name* and *count* are present, *name* takes precedence. *name* must meet the requirements for names as described in *name* must meet the requirements for names as described in Section 6.3, page 76.

The *expression* variables specify the expressions to be compared. All symbols in the expression must be defined previously. If an expression is not specified, the absolute value of 0 is used. *expressions* must meet the requirements for expressions as described in Section 6.9, page 94.

The condition variable specifies the relation to be satisfied by the two strings. You can enter *condition* in mixed case, and it must be one of the following:

- `LT` (less than)

  The value of the first expression must be less than the value of the second expression. The attributes are not checked.

- `LE` (less than or equal)

  The value of the first expression must be less than or equal to the value of the second expression. The attributes are not checked.

- GT (greater than)

  The value of the first expression must be greater than the value of the second expression. The attributes are not checked.

- GE (greater than or equal)

  The value of the first expression must be greater than or equal to the value of the second expression. The attributes are not checked.

- EQ (equal)

  The value of the first expression must be equal to the value of the second expression. Both expressions must be one of the following:

  - Attributes must be the same
  - Immobile relative to the same section
  - Relocatable relative to the same section
  - External relative to the same external symbol.
  - The word-address, byte-address, or value

- NE (not equal)

  The first expression and the second expression do not satisfy the conditions required for EQ described above.

The *count* variable specifies the statement count. It must be an absolute expression with a positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count. *count* is used only when the label field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

The following example illustrates the use of the IFE pseudo instruction:

```
        IDENT   TEST
SYM1    =       0
SYM2    =       *
SYM3    SET     1000
SYM4    SET     500
NOTEQ   IFE     SYM1,EQ,SYM2        ; Condition fails, values are the same,
                                    ; but the attributes are different.
        S1      SYM1                ; The ELSE sequence is assembled.
        S2      SYM2
```

```
NOTEQ   ELSE
        S1    SYM3              ; Statement is included.
        S2    SYM4              ; Statement is included.
NOTEQ   ENDIF                   ; End of conditional sequence.
        END
```

8.34 IFM

The IFM pseudo instruction tests characteristics of the current target machine. If the result of the machine condition is true, assembly continues with the next statement. If the result of the machine condition is false, subsequent statements are skipped. If a label field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same *name* is encountered; otherwise, skipping stops when the statement count is exhausted.

If any errors are encountered during the evaluation of the string condition, the resulting condition is handled as if true and an appropriate listing message is issued.

You can specify the IFM pseudo instruction anywhere within a program segment. If the IFM pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IFM pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the IFM pseudo instruction is as follows:

[*name*]	IFM	[# or !]*logical-name*[, [*count*]]
[*name*]	IFM	*numeric-name,condition*, [*expression*] [, [*count*]]

The *name* variable specifies an optional name of a conditional sequence of code. A conditional sequence of code that is controlled by a name is ended by an ENDIF pseudo instruction with a matching name. To reverse the condition of a conditional sequence of code controlled by a name, use an ELSE pseudo instruction with a matching name. If both *name* and *count* are present, *name* takes precedence. *name* must meet the requirements for names as described in Section 6.3, page 76.

The *logical-name* variable specifies the mnemonic that signifies a logical condition of the machine for which the assembler is currently targeting code. If the logical name is preceded by a pound sign (#) or an exclamation mark (!), its resulting condition is complemented. For a detailed list of the mnemonics, see the logical

traits of the CPU option for the appropriate operating system in Chapter 5, page 47.

The *numeric-name* variable specifies the mnemonic that signifies a numeric condition of the machine for which the assembler is currently targeting code. For a detailed list of the mnemonics, see the numeric traits of the CPU option for the appropriate operating system in Chapter 5, page 47. You can specify these mnemonics in mixed case.

The *condition* variable specifies the relation to be satisfied between the numeric name and the expression, if any. You can enter *condition* in mixed case, and it must be one of the following:

- LT (less than)

  The value of the numeric name must be less than the value of the expression.

- LE (less than or equal)

  The value of the numeric name must be less than or equal to the value of the expression.

- GT (greater than)

  The value of the numeric name must be greater than the value of the expression.

- GE (greater than or equal)

  The value of the numeric name must be greater than or equal to the value of the expression.

- EQ (equal)

  The value of the numeric name must be equal to the value of the expression.

- NE (not equal)

  The value of the numeric name must not equal the value of the expression.

The *expression* variable specifies the expression to be compared to the numeric name. All symbols in the expression must be defined previously and must have an address attribute of value and a relative attribute of absolute. If the current base is mixed, a default of decimal is used. If an expression is not specified, the absolute value of 0 is used. *expression* must meet the requirements for expressions as described in Section 6.9, page 94.

The *count* variable specifies the statement count. It must be an absolute expression with a positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count. *count* is used only when the label field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

The following example illustrates the use of the IFM pseudo instruction:

```
      ident  test
ex1   ifm    vpop              ; Assuming the condition is true,
      .                        ; skipping does occur within the IFM
      .                        ; part.
      .
ex1   ifm    numcpus,eq,4      ; Assuming the condition is false,
     .                         ; skipping occurs.
     .
     .
ex2  else                     ; Toggles the condition so that the else
     .                        ; part is not skipped.
     .
     .
ex2  endif
     end
```

8.35 INCLUDE

The INCLUDE pseudo instruction inserts a file at the current source position. The INCLUDE pseudo instruction always prepares the file for reading by opening it and positioning the pointer at the beginning.

You can use this pseudo instruction to include the same file more than once within a particular file.

You can also nest INCLUDE instructions. Because you cannot use INCLUDE recursively, you should review nested INCLUDE instructions for recursive calls to a file that you have already opened.

You can specify the INCLUDE pseudo instruction anywhere within a program segment. If the INCLUDE pseudo instruction occurs within a definition, it is recognized as a pseudo instruction and the specified file is included in the definition. If the INCLUDE pseudo instruction occurs within a skipping sequence, it is recognized as a pseudo instruction and the specified file is included in the

skipping sequence. The INCLUDE pseudo instruction statement itself is not inserted into a defined sequence of code.

Note: The INCLUDE pseudo instruction can be forced into a definition or skipped sequence of code. When editing is enabled, INCLUDE is expanded during execution and the file is read in at that point. This method is not recommended because formal parameters are not substituted correctly into statements when the INCLUDE macro is expanded during execution. If using this method, insert an underscore (_) anywhere within the pseudo instruction, as follows: IN_CLUDE. If editing is disabled during execution, INCLUDE is not expanded.

The format of the INCLUDE pseudo instruction is as follows:

```
ignored      INCLUDE      filename
```

The *filename* variable is an ASCII character string that identifies the file to be included. The ASCII character string must be a valid file name depending on the operating system under which the assembler is executing. If the ASCII character string is not a valid file name or the assembler cannot open the file, a listing message is issued.

filename must be specified with one matching character on each end. Any ASCII character other than a comma or space can be used. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

In the following examples, the module named INCTEST contains an INCLUDE pseudo instruction. The file to be included is named DOG and the CAT file is included within the DOG file.

The INCTEST module is as follows:

```
IDENT   INCTEST
INCLUDE  *DOG*      ; Call file DOG with INCLUDE.
END
```

The file DOG contains the following:

```
S1    1         ; Register S1 gets 1.
INCLUDE  'CAT'     ; Call file CAT with INCLUDE.
S2       2        ; Register S2 gets 2.
```

The file CAT contains the following:

```
S3    3       ; Register S3 gets 3.
```

The expansion of the INCTEST module is as follows:

```
IDENT   INCTEST
INCLUDE  *DOG*          ; Call file DOG with INCLUDE.
S1        1             ; Register S1 gets 1.
INCLUDE  'CAT'          ; Call file CAT with INCLUDE.
S3        3             ; Register S3 gets 3.
S2        2             ; Register S2 gets 2.
END
```

The following example demonstrates that it is illegal to include a file recursively within nested INCLUDE instructions.

The INCTEST module is as follows:

```
IDENT   INCTEST
INCLUDE  *DOG*          ; Call file DOG with INCLUDE.
END
```

The file DOG contains the following:

```
S1    1          ; Register S1 gets 1.
INCLUDE   'CAT'        ; Call file CAT with INCLUDE.
S2        2            ; Register S2 gets 2.
```

The file CAT includes the following:

```
S3     3   ; Register S3 gets 3.
INCLUDE   -DOG-    ; Illegal. If file B was included by
                   ; file A, it cannot include file A.
```

The following example demonstrates that it is legal to include a file more than once if it is not currently being included.

The INCTEST module is as follows:

```
IDENT   INCTEST
INCLUDE  *DOG*          ; Call file DOG with INCLUDE.
INCLUDE  *DOG*          ; Call file DOG with INCLUDE.
END
```

The file DOG contains the following:

```
S1    1           ; Register S1 gets 1.
S2    2           ; Register S2 gets 2.
```

The expansion of the INCTEST module is as follows:

```
IDENT   INCTEST
      INCLUDE  *DOG*    ; Call file DOG with INCLUDE.
      S1       1        ; Register S1 gets 1.
      S2       2        ; Register S2 gets 2.
      INCLUDE  *DOG*    ; Call file DOG with INCLUDE.
      S1       1        ; Register S1 gets 1.
      S2       2        ; Register S2 gets 2.
      END
```

8.36 LIST

The LIST pseudo instruction controls the listing. LIST is a list control pseudo instruction and by default, is not listed. To include the LIST pseudo instruction on the listing, specify the LIS option on this instruction. An END pseudo instruction resets options to the default values.

You can specify the LIST pseudo instruction anywhere within a program segment. If the LIST pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the LIST pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the LIST pseudo instruction is as follows:

```
[name]      LIST      [option]{ , [option]}*
```

The name variable specifies the optional list name. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

If *name* is present, the instruction is ignored unless a matching name is specified on the list parameter on the assembler invocation statement. LIST pseudo instructions with a matching name are not ignored. LIST pseudo instructions with a blank label field are always processed.

The *option* variable specifies that a particular listing feature be enabled or disabled. All option names can be specified in some form as assembler invocation statement parameters. The selection of an option on the assembler invocation statement overrides the enabling or disabling of the corresponding feature by

a LIST pseudo instruction. If you use the no-list option on the assembler invocation statement, all LIST pseudo instructions in the program are ignored.

There can be zero, one, or more options specified or an *. If no options are specified, OFF is assumed. The allowed options are described as follows:

- ON (enables source statement listing)

  Source statements and code generated are listed (default).

- OFF (disables source statement listing)

  While this option is selected, only statements with errors are listed. If the LIS option is enabled, listing control pseudo instructions are also listed.

- ED (enables listing of edited statements)

  Edited statements are included in the listing file (default).

- NED (disables listing of edited statements)

  Edited statements are not included in the listing file.

- XRF (enables cross-reference)

  Symbol references are accumulated and a cross-reference listing is produced (default).

- NXRF (disables cross-reference)

  Symbol references are not accumulated. If this option is selected when the END pseudo instruction is encountered, no cross-reference is produced.

- XNS (includes nonreferenced local symbols in the reference)

  Local symbols that were not referenced in the listing output are included in the cross-reference listing (default).

- NXNS (excludes nonreferenced local symbols from the cross-reference)

  If this option is selected when the END pseudo instruction is encountered, local symbols that were not referenced in the listing output are not included in the cross-reference.

- LIS (enables listing of the listing pseudo instructions)

  The LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, and ENDTEXT pseudo instructions are included in the listing.

- NLIS (disables listing of the listing pseudo instructions)

The LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, and ENDTEXT pseudo instructions are not included in the listing (default).

- TXT (enables global text source listing)

  Each statement following a TEXT pseudo instruction is listed through the ENDTEXT instruction if the listing is otherwise enabled.

- NTXT (disables global text source listing)

  Statements that follow a TEXT pseudo instruction through the following ENDTEXT instruction are not listed (default).

- MAC (enables listing of macro and opdef expansions)

  Statements generated by macro and opdef calls are listed. Conditional statements and skipped statements generated by macro and opdef calls are not listed unless the macro conditional list feature is enabled (MIF).

- NMAC (disables listing of macro and opdef expansions)

  Statements generated by macro and opdef calls are not listed (default).

- MBO (enables listing of generated statements before editing)

  Only statements that produce generated code are listed. The listing of macro expansions (MAC) or the listing of duplicated statements (DUP) must also be enabled.

- NMBO (disables listing of statements that produce generated code)

  Statements generated by a macro or opdef call (MAC), or by a DUP or ECHO (DUP) pseudo instruction, are not listed before editing (default).

  Note: Source statements containing a micro reference (see MIC and NMIC options) or a concatenation character are listed before editing regardless of whether this option is enabled or disabled.

- MIC (enables listing of generated statements before editing)

  Statements that are generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, and that contain a micro reference or concatenation character are listed before and after editing. The listing of macro expansions or the listing of duplicated statements must also be enabled.

- NMIC (disables listing of generated statements before editing)

  Statements generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, are not listed before editing (default).

Note: Conditional statements (see NIF and NMIF options) and skipped statements in source code are listed regardless of whether this option is enabled or disabled.

- MIF (enables macro conditional listing)

  Conditional statements and skipped statements generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, are listed. The listing of macro expansions or the listing of duplicated statements must also be enabled.

- NMIF (disables macro conditional listing)

  Conditional statements and skipped statements generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, are not listed (default).

- DUP (enables listing of duplicated statements)

  Statements generated by DUP and ECHO expansions are listed. Conditional statements and skipped statements generated by DUP and ECHO are not listed unless the macro conditional list feature is enabled (MIF).

- NDUP (disables listing of duplicated statements)

  Statements generated by DUP and ECHO are not listed (default).

The asterisk (*) reactivates the LIST pseudo instruction in effect before the current LIST pseudo instruction was specified within the current program segment. Each occurrence of a LIST pseudo instruction other than LIST initiates a new listing control. Each LIST releases the current listing control and reactivates the listing control that preceded the current list control. If all specified listing controls were released when a LIST * is encountered, the assembler issues a caution-level message and uses the defaults for listing control.

8.37 LOC

The LOC pseudo instruction sets the location counter to the first byte of the word address specified. The location counter is used for assigning address values to label field symbols. Changing the location counter allows code to be assembled and loaded at one location, controlled by the origin counter, then moved and executed at another address controlled by the location counter. The LOC pseudo instruction forces a word boundary within the current section before the location counter is modified.

The LOC pseudo instruction is restricted to sections that allow instructions or data, or both. If the LOC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the LOC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the LOC pseudo instruction is as follows:

```
ignored     LOC     [expression]
```

The *expression* variable is optional and represents the new value of the location counter. If the expression does not exist, the counter is reset to the absolute value of 0. If the expression does exist, all symbols (if any) must be defined previously. If the current base is mixed, octal is used as the base.

The *expression* operand cannot have an address attribute of byte, a relative attribute of external, or a negative value. A force word boundary occurs before the expression is evaluated. The *expression* operand must meet the requirements for an expression as described in Section 6.9, page 94.

The following example illustrates the use of the LOC pseudo instruction:

```
        ORG   W.*+1000
        LOC   200
LBL     A1    0
        .
        .
        .
        J     LBL
```

Note: In the preceding example, the code is generated and loaded at location W.*+1000 and the user must move it to absolute location 200 before execution.

8.38 LOCAL

The LOCAL pseudo instruction specifies unique character string replacements within a program segment that are defined only within the macro, opdef, dup, or echo definition. These character string replacements are known only in the macro, opdef, dup, or echo at expansion time. The most common usage of the LOCAL pseudo instruction is for defining symbols, but it is not restricted to the definition of symbols.

The LOCAL pseudo instruction is described in detail in Section 9.11, page 258.

8.39 MACRO

The MACRO pseudo instruction marks the beginning of a sequence of source program instructions saved by the assembler for inclusion in a program when called for by the macro name.

Macros are described in detail in Section 9.2, page 219.

8.40 MICRO

The MICRO pseudo instruction assigns a name to a character string. The assigned name can be redefined. You can reference and redefine a redefinable micro after its initial definition within a program segment. A micro defined with the MICRO pseudo instruction is discarded at the end of a module and cannot be referenced by any of the segments that follow.

You can specify the MICRO pseudo instruction anywhere within a program segment. If the MICRO pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the MICRO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the MICRO pseudo instruction is as follows:

```
name    MICRO    [string[, [exp][, [exp], [case]]]]
```

The *name* variable is required and is assigned to the character string in the operand field. It has redefinable attributes. If *name* was previously defined, the previous micro definition is lost. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

The *string* variable represents an optional character string that can include previously defined micros. If *string* is not specified, an empty string is used. A character string can be delimited by any character other than a space. Two consecutive occurrences of the delimiting character indicate a single such character (for example, a micro consisting of the single character * can be specified as `'*'` or `****`).

The *exp* variable represents optional expressions. The first expression must be an absolute expression that indicates the number of characters in the micro character string. All symbols, if any, must be previously defined. If the current base is mixed, decimal is used for the expression. The expressions must meet the requirements for expressions as described in Section 6.9, page 94.

The micro character string is terminated by the value of the first expression or the final apostrophe of the character string, whichever occurs first. If the first expression has a 0 or negative value, the string is considered empty. If the first expression is not specified, the full value of the character string is used. In this case, the string is terminated by the final apostrophe.

The second expression must be an absolute expression that indicates the micro string's starting character. All symbols, if any, must be defined previously. If the current base is mixed, decimal is used for the expression.

The starting character of the micro string begins with the character that is equal to the value of the second expression, or with the first character in the character string if the second expression is null or has a value of 1 or less.

The optional *case* variable denotes the way uppercase and lowercase characters are interpreted when they are read from *string*. Character conversion is restricted to the letter characters (`A - Z` and `a - z`) specified in *string*. You can specify *case* in uppercase, lowercase, or mixed case, and it must be one of the following:

- `MIXED` or `mixed`

  string is interpreted as entered and no case conversion occurs. This is the default.

- `UPPER` or `upper`

  All lowercase alphabetic characters in *string* are converted to their uppercase equivalents.

- `LOWER` or `lower`

  All uppercase alphabetic characters in *string* are converted to their lowercase equivalents.

The following example illustrates the use of the `MICRO` pseudo instruction:

```
MIC    MICRO          'THIS IS A MICRO STRING'
MIC2   MICRO          '"MIC"',1
MIC2; CAL has edited these lines    MICRO 'THIS IS A MICRO STRING',1
MIC3   MICRO          '"MIC2"'
MIC3; CAL has edited these lines   MICRO        'T'
MIC4   MICRO          '"MIC"',10   ; CALL TO MICRO MIC2.
MIC4; CAL has edited these lines  MICRO           'THIS IS A MICRO STRING',10
MIC5   MICRO          '"MIC4"'  MIC5
MICRO                 'THIS IS A '
MIC6   MICRO          '"MIC" ',5,11
MIC6; CAL has edited these lines  MICRO           'THIS IS A MICRO STRING',5,11
```

```
MIC7   MICRO          '"MIC6"'
MIC7; CAL has edited these lines  MICRO          'MICRO'
MIC8   MICRO          '"MIC"',11,5
MIC8; CAL has edited these lines  MICRO          'THIS IS A MICRO STRING',11,5
MIC9   MICRO          '"MIC8"'
MIC9; CAL has edited these lines  MICRO          ' IS A MICRO'
```

8.41 MICSIZE

The MICSIZE pseudo instruction defines the symbol in the label field as a symbol with an address attribute of value, a relative attribute of absolute, and a value equal to the number of characters in the micro string whose name is in the operand field. Another SET or MICSIZE instruction with the same symbol redefines the symbol to a new value.

You can specify the MICSIZE pseudo instruction anywhere within a program segment. If the MICSIZE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the MICSIZE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the MICSIZE pseudo instruction is as follows:

```
[symbol]    MICSIZE    name
```

The *symbol* variable specifies an optional unqualified symbol. *symbol* is implicitly qualified by the current qualifier. The label field can be blank. *symbol* must meet the requirement for a symbol as described in Section 6.2, page 70.

The name variable represents the name of a micro string that has been previously defined. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

8.42 MLEVEL

The MLEVEL pseudo instruction sets the message level for the output listing, the message listing, and the standard error file.

If the option accompanying the MLEVEL pseudo instruction is not valid, a diagnostic message is generated and MLEVEL is ignored.

You can specify the MLEVEL pseudo instruction anywhere within a program segment. If the MLEVEL pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the MLEVEL pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the MLEVEL pseudo instruction is as follows:

```
ignored     MLEVEL     [option]/*
```

The *option* variable specifies an optional message level. It can be entered in uppercase, lowercase, or mixed case, and it must be one of the following levels:

- ERROR (enables ERROR level messages only)
- WARNING (enables WARNING and ERROR level messages)
- CAUTION (enables CAUTION, WARNING, and ERROR level messages)
- NOTE (enables NOTE, CAUTION, WARNING, and ERROR level messages)
- COMMENT (enables COMMENT, NOTE, CAUTION, WARNING, and ERROR level messages)
- The asterisk (*) reactivates the message level in effect before the current message level was specified within the current program segment. Each occurrence of an MLEVEL pseudo instruction other than MLEVEL * initiates a new message level. Each MLEVEL * releases the current message level and reactivates the message level that preceded the current message level. If all specified message levels have been released when an MLEVEL * is encountered, the assembler issues a CAUTION message to alert you to the situation and then reverts to the default level, WARNING.
- If you do not specify a level, the level is reset to the default, which is WARNING.

8.43 MSG

The MSG pseudo instruction enables specified assembler messages to be printed. It essentially increases the severity of the specified messages so that when encountered, the message will be printed. The format is as follows:

```
        MSG     msgnum[:msgnum ... ]
```

msgnum	Number of the assembler message to print. Multiple message numbers are separated by colons.

If the value of the -M *message* option on the as statement differs from the option on the MSG pseudo instruction, the as statement value is used.

If the option accompanying the MSG pseudo instruction is not valid, a diagnostic message is generated, and MSG is ignored.

If MSG is placed in global scope (outside of any IDENT/END pair), it applies to the file. If MSG is placed in module scope (inside an IDENT/END pair), it applies to just that module. If there are multiple MSG and NOMSG (see Section 8.45, page 182) pseudo instructions for the same message number, the last applies throughout. The two pseudo instructions cannot be toggled.

8.44 NEXTDUP

The NEXTDUP pseudo instruction stops the current iteration of a duplication sequence indicated by a DUP or an ECHO pseudo instruction. Assembly of the current repetition of the dup sequence is terminated immediately and the next repetition, if any, is begun.

The NEXTDUP pseudo instruction is described in detail in Section 9.9, page 254.

8.45 NOMSG

The NOMSG pseudo instruction disables specified assembler messages to prevent them from being printed. It essentially decreases the severity of the messages so that when encountered, the message will not be printed. The format is as follows:

```
NOMSG    msgnum[:msgnum ... ]
```

msgnum	Number of the assembler message to disable. Multiple message numbers are separated by colons.

If the value of the -M *message* option on the as statement differs from the option on the NOMSG pseudo instruction, the as statement value is used.

If the option accompanying the NOMSG pseudo instruction is not valid, a diagnostic message is generated, and NOMSG is ignored.

If NOMSG is placed in global scope (outside of any IDENT/END pair), it applies to the file. If NOMSG is placed in module scope (inside an IDENT/END pair), it applies to just that module. If there are multiple NOMSG and MSG (see Section 8.43, page 181) pseudo instructions for the same message number, the last applies throughout. The two pseudo instructions cannot be toggled.

8.46 OCTMIC

The OCTMIC pseudo instruction converts the value of an expression to a character string that is assigned a redefinable micro name. The character string that the pseudo instruction generates is represented as an octal number. The final length of the micro string is inserted into the code field of the listing.

You can specify OCTMIC with zero, one, or two expressions. The value of the first expression is converted to a micro string with a character length equal to the second expression. If the second expression is not specified, the minimum number of characters needed to represent the octal value of the first expression is used.

If the second expression is specified, the string is equal to the length specified by the second expression. If the number of characters in the micro string is less than the value of the second expression, the character value is right justified with the specified fill characters (zeros or blanks) preceding the value. If the number of characters in the string is greater than the value of the second expression, the beginning characters of the string are truncated and a warning message is issued.

You can specify the OCTMIC pseudo instruction anywhere within a program segment. If the OCTMIC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the OCTMIC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the OCTMIC pseudo instruction is as follows:

name	OCTMIC	[*expression*$_1$] [", "[*expression*$_2$[", "[*option*]]]]

The *name* variable is required and specifies the name of the micro. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

The *expression*$_1$ variable is an optional expression and is equal to *name*. If specified, *expression*$_1$ must have an address attribute of value and a relative attribute of absolute. All symbols used must be previously defined. If the current

base is mixed, a default of octal is used. If the first expression is not specified, the absolute value of 0 is used in creating the micro string. The *expression*$_1$ operand must meet the requirements for expressions as described in Section 6.9, page 94.

expression$_2$ provides a positive character count less than or equal to decimal 22. If this parameter is present, leading zeros or blanks (depending on *option*) are supplied, if necessary, to provide the requested number of characters. If specified, *expression*$_2$ must have an address attribute of value and a relative attribute of absolute with all symbols, if any, previously defined. If the current base is mixed, a default of decimal is used. If *expression*$_2$ is not specified, the micro string is represented in the minimum number of characters needed to represent the octal value of the first expression. The *expression*$_2$ operand must meet the requirements for expressions as described in Section 6.9, page 94.

option represents the type of fill characters (ZERO for zeros or BLANK for spaces) that will be used if the second expression is present and fill is needed. The default is ZERO. You can enter *option* in mixed case.

The following example illustrates the use of the OCTMIC pseudo instruction:

```
     IDENT   EXOCT
     BASE    0                  ; The base is octal.
ONE  OCTMIC  1,2
_*   "ONE"                      ; Returns 1 in 2 digits.
*    01                         ; Returns 1 in 2 digits.
TWO OCTMIC   5*7+60+700,3
_*  "TWO"                      ; Returns 1023 in 3 digits.
*
*   023                        ; Returns 1023 in 3 digits.
*
THREE OCTMIC  256000,10,ZERO
_*  "THREE"                    ; Zero fill on the left.
*   00256000                   ; Zero fill on the left.
FOUR  OCTMIC  256000,10,BLANK
_*  " FOUR"                    ; Blank fill (^) on the left.
*
*   ^^256000                   ; Blank fill (^) on the left.
*
    END
```

8.47 HEXMIC

The HEXMIC pseudo instruction converts the value of an expression to a character string that is assigned a redefinable micro name. The character string that the

pseudo instruction generates is represented as an hexadecimal number. The final length of the micro string is inserted into the code field of the listing.

You can specify HEXMIC with zero, one, or two expressions. The value of the first expression is converted to a micro string with a character length equal to the second expression. If the second expression is not specified, the minimum number of characters needed to represent the hexadecimal value of the first expression is used.

If the second expression is specified, the string is equal to the length specified by the second expression. If the number of characters in the micro string is less than the value of the second expression, the character value is right justified with the specified fill characters (zeros or blanks) preceding the value. If the number of characters in the string is greater than the value of the second expression, the beginning characters of the string are truncated and a warning message is issued.

You can specify the HEXMIC pseudo instruction anywhere within a program segment. If the HEXMIC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the HEXMIC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the HEXMIC pseudo instruction is as follows:

name	HEXMIC	[*expression*$_1$] [", "[*expression*$_2$[", "[*option*]]]]

The *name* variable is required and specifies the name of the micro. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

The *expression*$_1$ variable is an optional expression and is equal to *name*. If specified, *expression*$_1$ must have an address attribute of value and a relative attribute of absolute. All symbols used must be previously defined. If the current base is mixed, a default of hexadecimal is used. If the first expression is not specified, the absolute value of 0 is used in creating the micro string. The *expression*$_1$ operand must meet the requirements for expressions as described in Section 6.9, page 94.

expression$_2$ provides a positive character count less than or equal to decimal 22. If this parameter is present, leading zeros or blanks (depending on *option*) are supplied, if necessary, to provide the requested number of characters. If specified, *expression*$_2$ must have an address attribute of value and a relative attribute of absolute with all symbols, if any, previously defined. If the current base is mixed, a default of decimal is used. If *expression*$_2$ is not specified, the micro string is represented in the minimum number of characters needed to represent the

hexadecimal value of the first expression. The *expression*$_2$ operand must meet the requirements for expressions as described in Section 6.9, page 94.

option represents the type of fill characters (ZERO for zeros or BLANK for spaces) that will be used if the second expression is present and fill is needed. The default is ZERO. You can enter *option* in mixed case.

The following example illustrates the use of the HEXMIC pseudo instruction:

```
     IDENT   EXOCT
     BASE    0                  ; The base is hexidecimal.
ONE  HEXMIC  1,2
_*   "ONE"                      ; Returns 1 in 2 digits.
*    01                         ; Returns 1 in 2 digits.
TWO HEXMIC   5*7+60+700,3
_*  "TWO"                      ; Returns 1023 in 3 digits.
*
*   023                        ; Returns 1023 in 3 digits.
*
THREE HEXMIC  256000,10,ZERO
_*  "THREE"                    ; Zero fill on the left.
*   00256000                   ; Zero fill on the left.
FOUR  HEXMIC  256000,10,BLANK
_*  " FOUR"                    ; Blank fill (^) on the left.
*
*   ^^256000                   ; Blank fill (^) on the left.
*
    END
```

8.48 OPDEF

The OPDEF pseudo instruction marks the beginning of an operation definition (opdef). The opdef identifies a sequence of statements to be called later in the source program by an opdef call. Each time the opdef call occurs, the definition sequence is placed into the source program.

The OPDEF pseudo instruction is described in detail in Section 9.3, page 235.

8.49 OPSYN

The OPSYN pseudo instruction defines an operation that is synonymous with another macro or pseudo instruction operation.

The OPSYN pseudo instruction is described in detail in Section 9.12, page 260.

8.50 ORG

The ORG pseudo instruction resets the location and origin counters to the byte address specified. ORG resets the location and origin counters to the same address relative to the same section.

The ORG pseudo instruction forces a 64-bit longword boundary within the current section and also within the new section specified by the expression. These forced longword boundaries occur before the counter is reset. ORG can change the current working section without modifying the section stack.

The ORG pseudo instruction is restricted to sections that allow instructions or data, or instructions and data. If the ORG pseudo instruction is found within a definition, it is defined and not recognized as a pseudo instruction. If the ORG pseudo instruction is found within a skipping sequence, it is skipped and not recognized as a pseudo instruction.

The format of the ORG pseudo instruction is as follows:

```
ignored     ORG     [expression]
```

The *expression* variable is an optional immobile or relocatable expression with positive relocation within the section currently in use. If the expression is blank, the longword address of the next available longword in the section is used. A force 64-bit longword boundary occurs before the expression is evaluated.

The expression must have a value or word-address attribute. If the expression has a value attribute, it is assumed to be a longword address. If the expression exists, all symbols (if any) must be defined previously. If the current base is mixed, hexadecimal is used as the base.

The expression cannot have a relative attribute of absolute or external, or a negative value. The *expression* operand must meet the requirements for an expression as described in Section 6.9, page 94.

The following example illustrates the use of the ORG pseudo instruction:

```
            ORG    W.*+0'200
```

8.51 OSINFO

The OSINFO pseudo instruction allows the assembler to pass specific information to the linker. The format is as follows:

```
OSINFO      keyword=value
```

The keywords and the valid values for each are as follows:

Keyword	Value	
TEXTPAGESIZE	0	System default
	1	System minimum
	2	System maximum
DATAPAGESIZ	0	System default
	1	System minimum
	2	System maximum
MINMSPWIDTH	Minimum acceptable MSP width. The valid values are 1 through 4.	
MAXMSPWIDTH	Maximum acceptable MSP width. The valid values are 1 through 4.	
MINMAXVL	Minimum acceptable max-VL. Valid values are between 1 and 64.	
MAXMAXVL	Maximum acceptable max-VL. Valid values are between 1 and 64.	
VADDRWIDTH	Virtual address width in bits. The value must be 48.	
NUMBARRIERS	Number of barriers required. The value must be 0.	
SVXMODEL	The value must be 2.	
PROGMODEL	The value must be 0.	
HANDLECPF	0	Does not install a handler for continuous page fault signals.
	Anything else	Does install a handler for continuous page fault signals.
HANDLEUVL	0	Does not install a handler for unaligned vector load signals.

	Anything else	Does install a handler for unaligned vector load signals.
HANDLEMTO	0	Does not install a handler for MSYNC timeout signals.
	Anything else	Does install a handler for MSYNC timeout signals.

The *value* is a constant or a symbol that resolves to a constant.

8.52 QUAL

A QUAL pseudo instruction begins or ends a code sequence in which all symbols defined are qualified by a qualifier specified by the QUAL pseudo instruction or are unqualified. Until the first use of a QUAL pseudo instruction, symbols are defined as unqualified for each program segment. Global symbols cannot be qualified. The QUAL pseudo instruction must not occur before an IDENT pseudo instruction.

A qualifier applies only to symbols. Names used for sections, conditional sequences, duplicated sequences, macros, micros, externals, formal parameters, and so on, are not affected.

You must specify the QUAL pseudo instruction from within a program module. If the QUAL pseudo instruction is found within a definition or skipping sequence, it is defined and is not recognized as a pseudo instruction.

At the end of each program segment, all qualified symbols are discarded.

The format of the QUAL pseudo instruction is as follows:

```
ignored     QUAL     */[name]
```

The *name* variable is optional and indicates whether symbols will be qualified or unqualified and, if qualified, indicates the qualifier to be used. The *name* operand must meet the requirements for names as described Section 6.3, page 76.

The *name* operand causes all symbols defined until the next QUAL pseudo instruction to be qualified. A qualified symbol can be referenced with or without the qualifier that is currently active. If the symbol is referenced while some other qualifier is active, the reference must be in the following form:

/qualifier/symbol

When a symbol is referenced without a qualifier, the assembler tries to find it in the currently active qualifier. If the qualified symbol is not defined within the current qualifier, the assembler tries to find it in the list of unqualified symbols. If both of these searches fail, the symbol is undefined.

An unqualified symbol can be referenced explicitly using the following form:

//symbol

If the operand field of the QUAL is empty, symbols are unqualified until the next occurrence of a QUAL pseudo instruction. An unqualified symbol can be referenced without qualification from any place in the program module, or in the case of global symbols, from any program segment assembled after the symbol definition.

An asterisk (*) resumes use of the qualifier in effect before the most recent qualification within the current program segment. Each occurrence of a QUAL other than a QUAL * causes the initiation of a new qualifier. Each QUAL * removes the current qualifier and activates the most recent prior qualification. If the QUAL * statement is encountered and all specified qualifiers are released, a caution-level message is issued and succeeding symbols are defined as being unqualified.

The following example illustrates the use of the QUAL pseudo instruction:

```
* Assembler default for symbols is unqualified.
ABC    =   1
; ABC is unqualified.
       QUAL QNAME1
; Symbol qualifier QNAME1
ABC    =   2
; ABC is qualified by QNAME1.
       J   XYZ
XYZ    S1  A2
; XYZ is qualified by QNAME1.
       .
       .
       .
       QUAL QNAME2
```

```
; Symbol qualifier QNAME2.
ABC    =   3
       J   /QNAME1/XYZ
           .
           .
           .
       QUAL *
; Resume the use of symbols qualified with

; qualifier QNAME1.
           .
           .
           .
       QUAL *
; Resume the use of unqualified symbols
           .
           .
           .
A      IFA  DEF,ABC
; Test whether ABC is defined.
B      IFA  DEF,/QNAME1/ABC
; Test if ABC is defined within qualifier

; QNAME1
C      IFA  DEF,/QNAME2/ABC
; Test if /QNAME2/ABC is defined within

; qualifier QNAME2.
       .
       .
       .
```

8.53 SECTION

The SECTION pseudo instruction establishes or resumes a section of code. The section can be common or local, depending on the options found in the operand field. Each section has its own location, origin, and bit-position counters.

You must specify the SECTION pseudo instruction from within a program module. If the SECTION pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SECTION

pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the SECTION pseudo instruction is as follows:

```
[lname]     SECTION     [type]["[location]]["[ENTRY]]

[lname]     SECTION     [location]"[type]["[ENTRY]]


[lname]     SECTION     [type]["[ENTRY]]["[location]]


[lname]     SECTION     [location]"[[ENTRY]]["[type]]

[lname]     SECTION     [ENTRY]["[location]]["[type]]

[lname]     SECTION     [ENTRY]["[type]]["[location]]

ignored     SECTION     *
```

The variables associated with the SECTION pseudo instruction are described as follows:

- *lname*

  The *lname* variable is optional and names the section. *lname* must meet the requirements for tags as described in Section 6.3, page 76.

- *type*

  The type variable specifies the type of section. It can be specified in uppercase, lowercase, or mixed case. *type* can be one of the following (for a description of local sections, see Section 8.53.1, page 198):

  - MIXED

    Defines a section that permits both instructions and data. MIXED is the default type for the main section initiated by the IDENT pseudo instruction. If *type* is not specified, MIXED is the default. The loader treats a MIXED section as a local section.

- CODE

Restricts a section to instructions only; data is not permitted. The loader treats a CODE section as a local section.

- DATA

Restricts a section to data only (CON, DATA, BSSZ, and so on); instructions are not permitted. The loader treats the DATA section as a local section.

- ZERODATA

Neither instructions nor data are allowed within this section. The loader treats a ZERODATA section as a local section. At load time, all space within a ZERODATA section is set to 0.

- CONST

Restricts a section to constants only (CON, DATA, BSSZ, and so on); instructions are not permitted. The loader treats the CONST section as a local section.

- STACK

Sets up a stack frame (designated memory area). Neither data nor instructions are allowed. All symbols that are defined using the location or origin counter and are relative to a section that has a type of STACK are assigned a relative attribute of immobile.

These symbols may be used as offsets into the STACK section itself. These sections are treated like other section types except relocation does not occur after assembly. Because relocation does not occur, sections with a type of stack are not passed to the loader.

Sections with a type of STACK conveniently indicate that symbols are relative to an execution-time stack frame and that their values correspond to an absolute location within the stack frame relative to the base of the stack frame. Symbols with stack attributes are indicated as such in the debug tables that the assembler produces.

- COMMON

Defines a common section that can be referenced by another program module. Instructions are not allowed.

Data cannot be defined in a COMMON section without a name (no name in label field); only storage reservation can be defined in an unnamed COMMON section. The label field that names a COMMON section cannot match the label

field name of a previously defined section with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM. If duplicate label field names are specified, an error level message is issued.

For a description of unnamed (blank) COMMON, see Section 8.53.5, page 200.

- DYNAMIC

Allocates an expandable common section at load time. DYNAMIC is a common section. Neither instructions nor data are permitted within a DYNAMIC section; only storage reservation can be defined in an unnamed DYNAMIC section. The label field that names a DYNAMIC section cannot match the label field name of a previously defined section with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM. If duplicate label field names are specified, an error-level message is issued.

For a description of blank DYNAMIC, see Section 8.53.5, page 200.

- ZEROCOM

Defines a common section that can be referenced by another program module. Neither instructions nor data are permitted within a ZEROCOM section; only storage reservation can be defined.

At load time, all uninitialized space within a ZEROCOM section is set to 0 . If a COMMON section with the same name contains the initialized text that was referenced by another module that will be loaded, portions of a ZEROCOM section can be explicitly initialized to values other than 0.

ZEROCOM must always be named. The label field that names a ZEROCOM section cannot match the label field name of a previously defined section with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM. If duplicate label field names are specified, an error level message is issued.

- TASKCOM

Defines a task common section. Neither instructions nor data are allowed at assembly time. At execution time, TASKCOM is set up and can be referenced by all subroutines that are local to a task. Data also can be inserted at execution time into a TASKCOM section by any subroutine that is executed within a single task.

When a section is defined with a type of TASKCOM, the assembler creates a symbol that is assigned the name in the label field of the SECTION pseudo instruction that defines the section. This symbol is not redefinable, has a value of 0, an address attribute of word, and a relative attribute that is relocatable relative to the section. The loader relocates this symbol, and it

is used as an offset into an execution time task common table. The word at which it points within this table contains the address of the base of the task common section in memory.

All symbols defined using the location or origin counter within a task common section are assigned a relative attribute of immobile. These symbols are treated like other symbols, but relocation does not occur after assembly. These symbols can be used as offsets into the task common section itself.

Sections with a type of TASKCOM indicate that their symbols are relative to an execution-time task common section, and their values correspond to an absolute location within the task common section relative to the beginning of the task common section. These values are indicated as such in the debug tables that the assembler produces. For a description of local sections, see Section 8.53.1, page 198.

TASKCOM must always be named. The label field that names a TASKCOM section cannot match the label field name of a previously defined section with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM . If duplicate label field names are specified, an error level message is issued.

> Note: Accessing data from a task common section is not as straightforward as accessing data directly from memory. For more information about task common, see the *Fortran Language Reference Manual, Volume 3.*

- ENTRY

  Sets a bit in the Program header to direct the loader to create an entry point at the same address as the first word of the section.

- *

  The *name, type,* and *location* of the section in control reverts to the *name, type,* and *location* of the section in effect before the current section was specified within the current program module. Each occurrence of a SECTION pseudo instruction other than SECTION * causes a section with the *name, type,* and *location* specified to be allocated. Each SECTION * releases the currently active section and reactivates the section that preceded the current section. If all specified sections were released when a SECTION * is encountered, the assembler issues a caution-level message and uses the main section.

  When *type* is not specified, MIXED is used by default.

If *type* is not specified, the default is MIXED for *type.* Because a module within a program segment is initialized without a name, these defaults, when acting together, force this initial section entry to become the current working section.

If the section name and attributes are previously defined, the SECTION pseudo instruction makes the previously defined section entry the current working section. If the section name and attributes are not defined, the SECTION pseudo instruction tries to create a new section with the name and attributes. The following restrictions apply when a new section is created:

- A section of the type TASKCOM, COMMON, ZEROCOM, and a section with a specified entry must always have a label field name.
- If a section with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM is being created for the first time, it must never have a name that matches a section that was created previously with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM.

The following example illustrates the use of the SECTION pseudo instruction:

```
     ident  exsect
; The Main section has by default a type of mixed.
     .
     .
     con    1
; Data and instructions are permitted in
     S1     1
; the Main section.
     .
     .
dsect  section data
; This section is defined with a name of
     .
; dsect and a type of data.
     .
     con    3
; Data is permitted in dsect.
     bszz   2
; Data is permitted in dsect.
     .
     .
     .
     S2   S3
; The assembler generates an error-level
     .
; message because instructions are illegal in a
     .
; section with a type of data.
```

```
      .
csect  section common
; This section is defined with a name of
      .
; csect and a type of common.
      .
      data  '12345678'
; Data is permitted in a named common section.
      S2    A1
; The assembler generates an error-level
      .
; message, because instructions are not permitted
      .
; in a common section.
      .
      section
; This section is unnamed and is assigned

; by default a type of mixed. When a section is

; specified without a name and a type, the main
; section becomes the current section.
      section *
; The current section reverts to the previous section
; in the stack buffer csect.
      section *
; The current section reverts to the previous section in the
; stack buffer dsect.
      con   2
; A memory location with a value of 2 is inserted into dsect.
      .
      .
      section *
; The current section reverts to the main
      .
; section.
      .
      .
dsect  section code
; The assembler considers this section specification unique
; and different from the previously defined section named
; dsect. Sections with types of mixed, code, data, and stack
; are treated as local sections that are specified with the
```

```
; same name therefore, are, considered unique if they
     .
; are specified with different types.
     .
     s1   s2            ; Instructions are permitted in dsect.
csect  section common    ; The current section reverts to the
                         ; section defined previously as csect.
                         ; When a section is specified with the
                         ; name, type, and location of a previously
     .                   ; defined section, the previously defined
     .                   ; section becomes the current section.
     .
     section *           ; The current section reverts to the main
     .                   ; section
     .
     .
     con   2             ; The assembler generates an error-level
     .                   ; message because data is not permitted in a
     .                   ; section with a type of code.
     .
     section *           ; This current section reverts to the main
     .                   ; section.
     .
     .
csect  section dynamic   ; The assembler generates an error-level
                         ; message, because the loader does not treat
                         ; sections with types of common, dynamic,
                         ; and taskcom as local sections Specifying
                         ; a section with a previously defined name
                         ; is illegal when the accompanying type
                         ; does not define a local section.
     end
```

8.53.1 Local Sections

A local section is a block of code that is usable only by the program module in which it resides. CAL uses three types of local sections:

- Main section
- Literals section
- Sections defined by the SECTION pseudo instruction

When a SECTION pseudo instruction is used, every SECTION type except COMMON, DYNAMIC, TASKCOM, and ZEROCOM is local. For more information about SECTION types, see the SECTION pseudo instruction in Section 8.53, page 191.

8.53.2 Main Sections

The main section is initiated by the IDENT pseudo instruction and is always the first section in a program module. This section is used for all local code other than that generated by the occurrence of a literal reference or code between two SECTION pseudo instructions.

Generally, sections may not have names but must be assigned types and locations. The default name of the main section is always empty. The defaults for type and location are MIXED and CM, respectively. For more information about the MIXED and CM section names, see the SECTION pseudo instruction in Section 8.53, page 191.

8.53.3 Literals Section

The first use of a literal value in an expression causes the assembler to store the data item in a literals section. Data is generated in the literals section implicitly by the occurrence of a literal. Explicit data generation or memory reservation is not allowed in the literals section. The assembler supports the literals section as a constant section. For more information about literals, see Section 6.6, page 85.

8.53.4 Sections Defined by the SECTION Pseudo Instruction

When a SECTION pseudo instruction is used, all code generated or memory reserved (other than literals) between occurrences of SECTION pseudo instructions is assigned to the designated section.

Until the first SECTION pseudo instruction is specified, the main section is used. If you specify the ORG pseudo instruction, an exception to these conditions can occur. Specifying the ORG pseudo instruction may cause the placement of code or memory reservations to be different from the currently specified working section.

The SECTION pseudo instruction is recommended for use because it has all of the same capabilities as the BLOCK and COMMON pseudo instructions.

When a section is released, the type and location of the previous section is used. When the number of sections released is equal to or greater than the number specified, CAL uses the defaults of the main section for type (MIXED) and location (CM).

A section with the same name, type, and location used in different areas of a program is recognized as the same section. For more information, see the SECTION pseudo instruction in Section 8.53, page 191.

8.53.5 Common Sections

When a SECTION pseudo instruction is used with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM, all code generated (other than literals) or memory reserved between occurrences of SECTION pseudo instructions is assigned to the designated common, dynamic, zero common, or task common section.

At program end, each common section is identified to the loader by its SECTION name and is available for reference by another program module. If you specify the ORG pseudo instruction, an exception to these conditions can occur. Specifying the ORG pseudo instruction may cause the placement of code or memory reservations to be different from the currently specified working section.

If a common section is specified, the identifier in the label field that names the section must be unique within the module in which it is defined. When a section is assigned a type (COMMON, DYNAMIC, ZEROCOM, or TASKCOM) that differs from the type of a previously defined section, it cannot be assigned the name of a previously defined section within the same module.

8.53.6 Section Stack Buffer

The assembler maintains a stack buffer that contains a list of the sections specified. Each time a SECTION pseudo instruction names a new section, the assembler adds the name of the section to the list and identifies the new section as the current section. You also can use the BLOCK and COMMON pseudo instructions to name sections.

The assembler remembers the order in which sections are specified. An entry is deleted from the list each time a SECTION pseudo instruction contains an asterisk (*). When an entry is deleted, the name, location, and type of the previously specified section is enabled.

The first section on the list is the last section that will be deleted from the list. If the program contains more SECTION * instructions than there are entries, the assembler uses the main section. (The BLOCK * and COMMON * instructions replace the current section with the most recent previous section that was specified by the BLOCK and COMMON pseudo instructions.)

For each section used in a program, the assembler maintains an origin counter, a location counter, and a bit position counter. When a section is first established or its use is resumed, the assembler uses the counters for that section.

The following example illustrates section specification and deletion and indicates the current section. The example includes the QUAL pseudo instruction. For a description of the QUAL pseudo instruction, see Section 8.52, page 189.

```
IDENT   STACK           ; The IDENT statement puts the first entry
                        ; on the list of qualifiers.  This entry
                        ; starts the symbol table for unqualified
                        ; symbols.
SYM1  =     1           ; SYM1 is relative to the main section.
      QUAL   QNAME1     ; Second entry on the list of qualifiers.
SYM2  =     2           ; SYM2 is the first entry in the symbol
                        ; table for QNAME1.
SNAME SECTION  MIXED    ; SNAME is the second entry on the list of
                        ; sections
      MLEVEL  ERROR     ; Reset message level to error eliminate
                        ; warning level messages.
SYM3  =       *         ; SYM3 is the second entry in the symbol
                        ; table for QNAME1 and is relative to the
                        ; SNAME section.
     MLEVEL   *         ; Reset message level to default in effect
                        ; before the MLEVEL specification.
     SECTION  *         ; SNAME is deleted from the list of
                        ; sections.
SYM4  =       4         ; SYM4 is the third entry in the symbol
                        ; table for QNAME1 and is relative to the
                        ; main section.
      QUAL    QNAME2    ; Third entry on the list of qualifiers.
SYM5  =       5         ; SYM5 is the first entry in the symbol
                        ; table for QNAME2.
SYM6  =   /QNAME1/SYM2  ; SYM6 gets SYM2 from the symbol table for
                        ; QNAME1 even though QNAME1 is not the
                        ; current qualifier in effect.
      QUAL    *         ; QNAME2 is removed as the current
                        ; qualifier name.
SYM7  =     6           ; SYM7 is the fourth entry in the symbol
                        ; table for QNAME1.
SYM8  =     7           ; Second entry in the symbol table for
                        ; unqualified symbols.
```

8.53.7 Generated Code Position Counters

This subsection describes the generated code position counters.

8.53.7.1 Origin Counter

The *origin counter* controls the relative location of the next word that will be assembled or reserved in the section. You can reserve blank memory areas by using either the ORG or BSS pseudo instructions to advance the origin counter.

When the special element `*O` is used in an expression, the assembler replaces it with the current byte-address value of the origin counter for the section in use. To obtain the word-address value of the origin counter, use `W.*O`. (In this context, a word is a 64-bit long word.) For more information about the special elements and `W.` prefix, see Section 6.8, page 92.

8.53.7.2 Location Counter

Usually, the *location counter* is the same value as the origin counter and the assembler uses it to define symbolic addresses within a section. The counter is incremented when the origin counter is incremented. Use the LOC pseudo instruction to adjust the location counter so that it differs in value from the origin counter or so that it refers to the address relative to a section other than the one currently in use. When the special element `*` is used in an expression, the assembler replaces it with the current byte-address value of the location counter for the section in use. To obtain the 64-bit word-address value of the location counter, Use `W.*` (see Section 6.8, page 92).

8.53.7.3 Word-bit-position Counter

As instructions and data are assembled and placed into a 64-bit long word, the assembler maintains a pointer that indicates the next available bit within the word currently being assembled. This pointer is known as the *word-bit-position counter*. It is 0 at the beginning of a new word and is incremented by 1 for each completed bit in the word. Its maximum value is 63 for the rightmost bit in the word. When a word is completed, the origin and location counters are incremented by 1, and the word-bit-position counter is reset to 0 for the next word.

When the special element `*W` is used in an expression, the assembler replaces it with the current value of the word-bit-position counter. The normal advancement of the word-bit-position counter is in increments of 32 as instructions are generated. You can alter this normal advancement by using the BITW, BITP, DATA, and VWD pseudo instructions.

8.53.7.4 Force Word Boundary

If either of the following conditions are true, the assembler completes a partial word and sets the word-bit-position counter to 0:

- The current instruction is an ALIGN, BSS, BSSZ, CON, LOC, or ORG pseudo instruction.
- The current instruction is a DATA or VWD pseudo instruction and the instruction has an entry in the label field.

8.54 SET

The SET pseudo instruction resembles the = pseudo instruction; however, a symbol defined by SET is redefinable.

You can specify the SET pseudo instruction anywhere within a program segment. If the SET pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SET pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the SET pseudo instruction is as follows:

[*symbol*]	SET	*expression*[, [*attribute*]]

The *symbol* variable specifies an optional unqualified symbol. The symbol is implicitly qualified by the current qualifier. A symbol defined with the SET pseudo instruction can be redefined with another SET pseudo instruction, but the symbol must not be defined prior to the first SET pseudo instruction. The label field can be blank. *symbol* must meet the requirements for symbols as described in Section 6.2, page 70.

All symbols found within *expression* must have been previously defined. The *expression* operand must meet the requirements for an expression as described in Section 6.9, page 94.

The *attribute* variable specifies a byte (B), word (W), or value (V) attribute. Attribute, if present, is used rather than the expression's attribute. If a byte-address attribute is specified, an expression with word-address attribute is multiplied by four; if word-address attribute is specified, an expression with byte-address attribute is divided by four. An immobile or relocatable expression cannot be specified as having a value attribute.

The following example illustrates the use of the SET pseudo instruction:

```
SIZE    =    o'100
PARAM   SET  D'18
WORD    SET  *W
BYTE    SET  *B
SIZE    =    SIZE+1     ; Illegal
PARAM   SET  PARAM+2    ; Legal
```

8.55 SKIP

The SKIP pseudo instruction unconditionally skips subsequent statements. If a label field name is present, skipping stops when an ENDIF or ELSE with the same name is encountered; otherwise, skipping stops when the statement count is exhausted.

You can specify the SKIP pseudo instruction anywhere within a program segment. If the SKIP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SKIP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the SKIP pseudo instruction is as follows:

[*name*]	SKIP	[*count*]

The *name* variable specifies an optional name for a conditional sequence of code. If both *name* and *count* are present, *name* takes precedence. *name* must meet the requirements for identifiers as described in Section 6.3, page 76.

The *count* variable specifies a statement count. It must be an absolute expression with a positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count. *count* is used only when the label field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

The following example illustrates the use of the SKIP pseudo instruction:

```
         SKIP          ; No skipping occurs.
SNAME1   SKIP          ; Statements are skipped if an ENDIF or
         .             ; ELSE with a matching label field
         .             ; label is found.
```

```
          .
SNAME1    ENDIF
          .
          .
          .
SNAME2    SKIP 10        ; Statements are skipped until an ENDIF
                         ; or ELSE with a matching label field
          .              ; label is found.
          .
SNAME2    ENDIF
          .
          .
          .
          SKIP 4         ; Four statements are skipped.
```

8.56 SPACE

The SPACE pseudo instruction inserts the number of blank lines specified into the output listing. SPACE is a list control pseudo instruction and by default, is not listed. To include the SPACE pseudo instruction on the listing, specify the LIS option on the LIST pseudo instruction.

You can specify the SPACE pseudo instruction anywhere within a program segment. If the SPACE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SPACE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the SPACE pseudo instruction is as follows:

```
ignored      SPACE      [expression]
```

The expression variable specifies an optional absolute expression that specifies the number of blank lines to insert in the listing. *expression* must have an address attribute of value, a relative attribute of absolute, and a value of 0 or greater.

If *expression* is not specified, the absolute value of 1 is used and one blank line is inserted into the output listing. If the current base is mixed, a default of decimal is used for the expression.

The *expression* operand must meet the requirement for an expression as described in Section 6.9, page 94.

8.57 STACK

The STACK pseudo instruction increases the size of the stack. Increments made by the STACK pseudo instruction are cumulative. Each time the STACK pseudo instruction is used within a module, the current stack size is incremented by the number of words specified by the expression in the operand field of the STACK pseudo instruction.

The STACK pseudo instruction is used in conjunction with sections that have a type of STACK. If either a STACK section or the STACK pseudo instruction is specified within a module, the loader tables that the assembler produces indicate that the module uses one or more stacks. The stack size indicated in the loader tables is the combined sizes of all STACK sections, if any, added to the total value of all STACK pseudo instructions, if any, specified within a module.

You must specify the STACK pseudo instruction from within a program module. If the STACK pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the STACK pseudo instruction is found within a skipping sequence, it is skipped and not recognized as a pseudo instruction.

The format of the STACK pseudo instruction is as follows:

```
ignored     STACK     [expression]
```

The *expression* variable is optional. If specified, it must have an address attribute of word or value, a relative attribute of absolute, a positive value, and all symbols within it (if any) must be defined previously.

If STACK is specified without *expression*, the stack is not incremented. The *expression* operand must meet the requirements for an expression as described in Section 6.9, page 94.

8.58 START

The START pseudo instruction specifies the main program entry. The program uses the START pseudo instruction to specify the symbolic address at which execution begins following the loading of the program. The named symbol can optionally be an entry symbol specified in an ENTRY pseudo instruction.

You must specify the START pseudo instruction from within a program module. If the START pseudo instruction is found within a definition or skipping sequence, it is defined and is not recognized as a pseudo instruction.

The format of the START pseudo instruction is as follows:

```
ignored     START     symbol
```

The *symbol* variable must be the name of a symbol that is defined as an unqualified symbol within the same program module. *symbol* must not be redefinable, must have a relative attribute of relocatable, and cannot be relocatable relative to any section other than a section that allows instructions or a section that allows instructions and data. The START pseudo instruction cannot be specified in a section with a type of data only.

The length of the symbol is restricted depending on the type of loader table that the assembler is currently generating. If the symbol is too long, an error message results.

The *symbol* operand must meet the requirements for symbols as described Section 6.2, page 70.

The following example illustrates the use of the START pseudo instruction:

```
        IDENT   EXAMPLE
        START   HERE
HERE    =       *
        .
        .
        .
        END
```

8.59 STOPDUP

The STOPDUP pseudo instruction stops duplication of a code sequence indicated by a DUP or ECHO pseudo instruction.

The STOPDUP pseudo instruction is described in detail in Section 9.10, page 255.

8.60 SUBTITLE

The SUBTITLE pseudo instruction specifies the subtitle that will be printed on the listing. The instruction also causes a page eject. SUBTITLE is a list control pseudo instruction and is, by default, not listed. To include the SUBTITLE pseudo instruction on the listing, specify the LIS option on the LIST pseudo instruction.

You can specify the SUBTITLE pseudo instruction anywhere within a program segment. If the SUBTITLE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SUBTITLE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the SUBTITLE pseudo instruction is as follows:

```
ignored     SUBTITLE     [del-char[string-of-ASCII]del-char]
```

The *del-char* variable is the delimiting character. It must be a single matching character on both ends of the ASCII character string. Apostrophes and spaces are not legal delimiters; all other ASCII characters are allowed. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The *string-of-ASCII* variable is an ASCII character string that will be printed as the subtitle on subsequent pages of the listing. This string replaces any previous string found within the subtitle field.

8.61 TEXT

Source lines that follow the TEXT pseudo instruction through the next ENDTEXT pseudo instruction are treated as *text* source statements. These statements are listed only when the TXT listing option is enabled. A symbol defined in *text* source is treated as a text symbol for cross-reference purposes; that is, such a symbol is not listed in the cross-reference unless a reference to the symbol from a listed statement exists. The text *name* part of the cross-reference listing contains the text name.

If the text appears in the global part of a program segment, Symbols defined in *text* source are global. If the text appears within a program module, symbols in *text* source are local.

TEXT is a list control pseudo instruction and is, by default, not listed. The TEXT pseudo instruction is listed if the listing is on or if the LIS listing option is enabled regardless of other listing options.

The TEXT and ENDTEXT pseudo instructions have no effect on a binary definition file.

You can specify the TEXT pseudo instruction anywhere within a program segment. If the TEXT pseudo instruction is found within a definition, it is defined

and is not recognized as a pseudo instruction. If the TEXT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the TEXT pseudo instruction is as follows:

```
[name]      TEXT      [del-char[string-of-ASCII]del-char]
```

The *name* variable is optional. It is used as the name of the following source until the next ENDTEXT pseudo instruction. The name found in the label field is the text name for all defined symbols in the section, and it is listed in the text name part of the cross-reference listing.

The *name* location must meet the requirements for names as described in Section 6.3, page 76.

The *del-char* variable is the delimiting character. It must be a single matching character on both ends of the ASCII character string. Apostrophes and spaces are not legal delimiters; all other ASCII characters are allowed. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The *string-of-ASCII* variable is an ASCII character string that will be printed as the subtitle on subsequent pages of the listing. A maximum of 72 characters is allowed. This string replaces any previous string found within the subtitle field.

8.62 TITLE

The TITLE pseudo instruction specifies the main title that will be printed on the listing. TITLE is a list control pseudo instruction and is, by default, not listed. To include the TITLE pseudo instruction on the listing, specify the LIS option on the LIST pseudo instruction.

You can specify the TITLE pseudo instruction anywhere within a program segment. If the TITLE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the TITLE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the TITLE pseudo instruction is as follows:

```
ignored     TITLE     [del-char[string-of-ASCII]del-char]
```

The *del-char* variable is the delimiting character. It must be a single matching character on both ends of the ASCII character string. Apostrophes and spaces are not legal delimiters; all other ASCII characters are allowed. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The *string-of-ASCII* variable is an ASCII character string that will be printed to the diagnostic file. A maximum of 72 characters is allowed.

8.63 VWD

The VWD pseudo instruction allows data to be generated in fields that are from 0 to 64 bits wide. Fields can cross Longword boundaries. Data begins at the current bit position unless a symbol is used in the label field. If a symbol is present within the label field, a forced Longword boundary occurs, and the data begins at the new current bit position.

Code for each subfield is packed tightly with no unused bits inserted.

The VWD pseudo instruction is restricted to sections that have a type of instructions, data, or both. If the VWD pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the VWD pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the VWD pseudo instruction is as follows:

```
[symbol]    VWD    [count/[expression]][, [count/[expression]]]
```

The *symbol* variable represents an optional symbol. If *symbol* is present, a force word boundary occurs. The symbol is defined with the value of the location counter after the force word boundary and has an address attribute of word. *symbol* must meet the requirements for symbols as described in Section 6.2, page 70.

The *count* variable specifies the number of bits in the field. It can be a numeric constant or symbol with absolute and value attributes. *count* must be positive and less than or equal to 64. If a symbol is specified for *count*, it must have been previously defined. If one or more *count* entries are not valid, no code is generated for the entire set of subfields in the operand field; however, each subfield is still evaluated.

The *expression* variable represents the expression whose value will be inserted in the field. If *expression* is missing, the absolute value of 0 is used. If *count* is not equal to 0, the count is the number of bits reserved to store the following expression, if any. *expression* must meet the requirement for expressions as described in Section 6.9, page 94.

The following example illustrates the use of the VWD pseudo instruction:

```
      BASE  M
PDT   BSS   0
      VWD   1/SIGN,3/0,60A'"NAM"'R
                              ; 100000000000023440515
                              ; 10000000653
REMDR =     64-*W             ; 41
      VWD    REMDR/DSN        ; 00011044516
```

In the preceding example, the value of SIGN is 1, the value of FC is 0, the value of ADD is 653 (octal), and the value of DSN is $IN in ASCII code.

CAL Defined Sequences [9]

Defined sequences are named sequences of instructions that can be saved for assembly later in the source program. Defined sequences have several functional similarities.

The four types of defined sequences are specified by the MACRO, OPDEF, DUP, and ECHO pseudo instructions. The ENDM, ENDDUP, and STOPDUP pseudo instructions terminate defined sequences. The LOCAL and OPSYN pseudo instructions are associated with definitions and are included in this chapter.

The defined sequence pseudo instructions are as follows:

Pseudo	Description
MACRO	A sequence of source program instructions saved by the assembler for inclusion in a program when called for by the macro name. The macro call resembles a pseudo instruction.
OPDEF	A sequence of source program instructions saved by the assembler for inclusion in a program called for by the OPDEF pseudo instruction. The opdef resembles a symbolic machine instruction.
DUP	Introduces a sequence of code that is assembled repetitively a specified number of times; the duplicated code immediately follows the DUP pseudo instruction.
ECHO	Introduces a sequence of code that is assembled repetitively until an argument list is exhausted.
ENDM	Ends a macro or opdef definition.
ENDDUP	Terminates a DUP or ECHO sequence of code.
STOPDUP	Stops the duplication of a code sequence by overriding the repetition condition.
LOCAL	Specifies unique strings that are usually used as symbols within a MACRO, OPDEF, DUP, or ECHO pseudo instruction.
OPSYN	Defines a label field name that is the same as a specified operation in the operand field name.
EXITM	Terminates the innermost nested MACRO or OPDEF expansion.

9.1 Similarities Among Defined Sequences

Defined sequences have the following functional similarities:

- Editing
- Definition format
- Formal parameters
- Instruction calls
- Interact with the INCLUDE pseudo instruction

9.1.1 Editing

Assembler editing is disabled at definition time. The body of the definition (see Section 9.1.2, page 215) is saved before micros and concatenation marks are edited.

If editing is enabled, editing of the definition occurs during assembly each time it is called. The ENDDUP, ENDM, END, INCLUDE, and LOCAL pseudo instructions and prototype statements should not contain micros or concatenation characters because they may not be recognized at definition time.

When a sequence is defined, editing is disabled and cannot be explicitly enabled. When a sequence is called, the assembler performs the following operations:

- Checks all parameter substitutions marked at definition time
- Edits the statement if editing is enabled
- Processes the statement

By disabling editing at definition time (default) and specifying the INCLUDE pseudo instruction with embedded underscores, a saving in program overhead is achieved. Because editing is disabled at definition time, concatenation does not occur until the macro is called. If editing is enabled when the macro is called, the file is included at that time. This technique is demonstrated in the following example:

```
MACRO
INC
.
.
.
IN_CLUDE MYFILE    ; INCLUDE pseudo instruction with an embedded
```

```
.                    ; underscore
.
.
ENDM
```

Embedding underscores in an INCLUDE pseudo instruction becomes desirable when the INCLUDE pseudo instruction identifies large files. Because files are included when the macro is called and not at definition time, embedding underscores in the INCLUDE pseudo instruction can reduce the overhead required for a program.

9.1.2 Definition Format

MACRO, OPDEF, DUP, and ECHO pseudo instructions use the same definition format. The format consists of a header, body, and end.

The header consists of a MACRO, OPDEF, DUP, or ECHO pseudo instruction, a prototype statement for a MACRO or OPDEF definition, and, optionally, LOCAL pseudo instructions. For a macro, the prototype statement provides a macro functional definition and a list of formal parameters. For an opdef, the prototype statement supplies the syntax and the formal parameters.

LOCAL pseudo instructions identify parameter names that the assembler must make unique to the assembly each time the definition sequence is placed in a program segment. Asterisk comments can be placed in the header and do not affect the way the assembler scans the header. Asterisk comments are dropped from the definition. To force asterisk comments into a definition, see Section 6.10.5, page 100.

The body of the definition begins with the first statement following the header. The body can consist of a series of CAL instructions other than an END pseudo instruction. The body of a definition can be empty, or it can include other definitions and calls. A definition used within another definition is not recognized, however, until the definition in which it is contained is called; therefore, an inner definition cannot be called before the outer definition is called for the first time.

A comment statement identified by an asterisk in column 1 is ignored in the definition header and the definition body. Such comments are not saved as a part of the definition sequence. Comment fields on other statements in the body of a definition are saved.

An ENDM pseudo instruction with the proper name in the label field ends a macro or opdef definition. A statement count or an ENDDUP pseudo instruction with

the proper name in the label field ends a dup definition. An ENDDUP pseudo instruction with the proper name in the label field ends an echo definition.

9.1.3 Formal Parameters

Formal parameters are defined in the definition header and recognized in the definition body. Four types of formal parameters are recognized as follows:

- Positional
- Keyword
- Echo
- Local

The characters that identify positional, keyword, echo, and local parameters must all have unique names within a given definition. Positional, keyword, and echo parameters are also case-sensitive. To be recognized, you must specify these parameters in the body of the definition exactly as specified in the definition header. Parameter names must meet the requirements for identifiers as described in Section 6.3, page 76.

You can embed a formal parameter name within the definition body; however, embedded parameters must satisfy the following requirements:

- The first character of an embedded parameter must begin with a legal initial-identifier-character.
- An embedded parameter cannot be preceded by an initial-identifier-character (for example, PARAM is a legally embedded parameter within the ABC_PARAM_DEF string because it is preceded by an underscore character). PARAM is not a legally embedded character within the string ABCPARAMDEF because it is preceded by an initial-identifier-character (C).
- An embedded parameter must not be followed by an identifier-character.

  In the following example, the embedded parameter is legal because it is followed by an element separator (blank character):

  ```
  PARAM678
  ```

  In the following example, the embedded parameter is illegal because it is followed by the identifier-character 9:

  ```
  PARAM6789
  ```

- Embedded parameters must contain 8 or less characters. PARAM6789 is illegal because it contains 9 characters. The character that follows an embedded parameter (9) cannot be an identifier-character.
- If and only if the new format is specified, an embedded parameter must occur before the first comment character (;) of each statement within the body.
- An embedded parameter must have a matching formal parameter name in the definition header.
- Formal parameter names should not be END, ENDM, ENDDUP, LOCAL, or INCLUDE pseudo instructions. If any of these are used as parameter names, substitution of actual arguments occurs when these names are contained in any inner definition reference.

  Note: If the file is included at expansion time, arguments are not substituted for formal parameters into statements within included files.

9.1.4 Instruction Calls

Each time a definition sequence of code is called, an entry is added to a list of currently active defined sequences within the assembler. The most recent entry indicates the current source of statements to be assembled. When a definition is called within a definition sequence that is being assembled, another entry is made to the list of defined sequences, and assembly continues with the new definition sequence belonging to the inner, or nested, call.

At the end of a definition sequence, the most recent list entry is removed and assembly continues with the previous list entry. When the list of defined sequences is exhausted, assembly continues with statements from the source file.

An inner nested call can be recursive; that is, it can reference the same definition that is referenced by an outer call. The depth of nested calls permitted by CAL is limited only by the amount of memory available.

The sequence field in the right margin of the listing shows the definition name and nesting depth for defined sequences being assembled. Nesting depth numbers begin in column 89 and can be one of the following: `:1`, `:2`, `:3`, `:4`, `:5`, `:6`, `:7`, `:8`, `:9`, `:*`.

If the nesting depth is greater than 9, the assembler keeps track of the current nesting level; an asterisk represents nesting depths of 10 or more. Nesting depth numbers are restricted to two characters so that only the two rightmost character positions are overwritten.

If the sequence field (columns 73 through 90) of the source file is not empty, the assembler copies the existing field for a call into every statement expanded by the call reserving columns 89 and 90 for the nesting level. For example, if the sequence field for MCALL was LQ5992A.112, the sequence field for a statement expanded from MCALL would read as follows:

```
LQ5992A.112         :1
```

Additional nested calls within MCALL would change the nesting level, but the sequence field would be unchanged during MCALL. For example:

```
LQ5992A.112         :2
LQ5992A.112         :2
LQ5992A.112         :2
LQ5992A.112         :3
LQ5992A.112         :*
LQ5992A.112         :1
```

If the sequence field (columns 73 through 90) of the source file is empty, the assembler inserts the name of the definition, as follows:

Name	Description
Macro	The inserted name in the sequence field is the name found in the result field of the macro prototype statement.
Opdef	The inserted name in the sequence field is the name used in the label field of the OPDEF pseudo instruction itself.
Dup	The inserted name in the sequence field is the name used in the label field of the DUP pseudo instruction, or if the count is specified and name is not, the name is *Dup.
Echo	The inserted name in the sequence field is the name used in the label field of the ECHO pseudo instruction.

In all cases, the first two columns of the sequence field contain asterisks (**) to indicate that the assembler has generated the sequence field. Columns 89 and 90 of the sequence field are reserved for the nesting level. If, for example, the sequence field is missing for MCALL, it would read as follows:

```
** MCALL           :1
```

Additional nested calls within MCALL would change the nesting level, but the sequence field number would be unchanged for the duration of MCALL.

The following example illustrates how the assembler tracks the nesting sequence:

```
*MCALL          :1
*MCALL          :2
*MCALL          :2
*MCALL          :2
*MCALL          :3
** MCALL        :*
** MCALL        :1
```

9.1.5 Interaction with the INCLUDE Pseudo Instruction

The INCLUDE pseudo instruction operates with defined sequences, as follows:

Sequence	Description
MACRO	INCLUDE pseudo instructions are expanded at definition time.
OPDEF	INCLUDE pseudo instructions are expanded at definition time.
DUP	INCLUDE pseudo instructions are expanded at definition time. If count is specified, the INCLUDE pseudo instruction statement itself is not included in the statements being counted.
ECHO	INCLUDE pseudo instructions are expanded at definition time.

9.2 Macros (MACRO)

A macro definition identifies a sequence of statements. This sequence of statements is saved by the assembler for inclusion elsewhere in a program. A macro is referenced later in the source program by the *macro call.* Each time the macro call occurs, the definition sequence is placed into the source program.

You can specify the MACRO pseudo instruction anywhere within a program segment. If the MACRO pseudo instruction is found within a definition, it is defined. If the MACRO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

If a macro definition occurs within the global definitions part of a program segment, it is defined as global. If macro definitions occur within a program module (an IDENT, END sequence), they are local. A global definition can be redefined locally, however, at the end of the program module, it is re-enabled and the local definition is discarded. A global definition can be referenced from anywhere within the assembler program following the definition.

The following example illustrates a macro definition:

```
        MACRO
```

```
        GLOBAL                  ; Globally defined.
* GLOBAL DEFINITION IS USED.
GLOBAL ENDM
        LIST   MAC
        GLOBAL                  ; Call to global definition.
* GLOBAL DEFINITION IS USED.
        IDENT  TEST
        GLOBAL                  ; Call to global definition.
* GLOBAL DEFINITION IS USED.
        MACRO                   ; Locally defined.
        GLOBAL                  ; Attempted global definition.
* Redefinition warning message is issued
* LOCAL DEFINITION IS USED.
GLOBAL ENDM
        GLOBAL                  ; Call to local definition.
* LOCAL DEFINITION IS USED.
        END                     ; Local definitions discarded
        IDENT  TEST2
        GLOBAL                  ; Call to global definition.
* GLOBAL DEFINITION IS USED.
        END
```

9.2.1 Macro Definition

The macro definition header consists of the MACRO pseudo instruction, a prototype statement, and optional LOCAL pseudo instructions. The prototype statement provides a name for the macro and a list of formal parameters and default arguments.

A comment statement, identified by an asterisk in column 1, is ignored in the definition header or definition body. Such comments are not saved as a part of the definition sequence. Comment fields on other statements in the body of a definition are saved.

The end of a macro definition is signaled by an ENDM pseudo instruction with a name that matches the name in the result field of the macro prototype statement. For a description of the ENDM pseudo instruction, see Section 9.6, page 252.

The following macro definition transfers an integer from an A register to an S register and converts it to a normalized floating-point number:

```
        macro
        intconv     p1,p2 ; p1=A reg, p2=S reg.
        p2          +f_p1 ; Transfer with special exp and sign
```

```
                                ; extension.
          p2              +f_p1 ; Normalize the S register.
intconv   endm                  ; End of macro definition.
```

As with every macro, INTCONV begins with the MACRO pseudo instruction. The second statement is the prototype statement, which names the macro and defines the parameters. The next three statements are definition statements that identify what the macro does. The ENDM pseudo instruction ends the macro definition.

The format of the macro definition is as follows:

```
ignored      MACRO        ignored
[location]   functional   parameters
             LOCAL        [[name], [name]]
             .
             .
             .
functional   ENDM
```

The variables in the above macro definition are described as follows:

- *location*

  The *location* variable specifies an optional label field parameter. It must be terminated by a space and it must meet the requirements for names as described in Section 6.3, page 76.

- *functional*

  The *functional* variable specifies the name of the macro. It must be a valid identifier or the equal sign. If *functional* is the same as a currently defined pseudo instruction or macro, this definition redefines the operation associated with *functional*, and a message is issued.

- *parameters*

  The *parameters* variable specifies *positional* and/or *keyword* parameters. Positional parameters must be entered before keyword parameters. Keyword

parameters do not have to follow positional parameters. The syntax of the *parameter* variable is as follows:

```
positional-parameters[, [keyword-parameters]]
```

The syntax for *positional-parameters* is described as follows:

```
[[|] [*]name] [, positional-parameters]
```

The variables that comprise the positional parameter are described as follows:

- |/*

The vertical bar (|) is optional. If it is not included, the *positional-parameter's* argument can be an embedded argument, character string, or null string. If the vertical bar (|) is included, the parameter can be a syntactically valid expression or a null string.

A left parenthesis signals the beginning of an *embedded argument* and must be terminated by a matching right parenthesis. An embedded argument can contain an argument or pairs of matching left and right parentheses. If an asterisk precedes the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not precede the positional parameter name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

A *character string* can be any character up to but not including a legal terminator (space, tab, or semicolon for new format) or an element separator (comma). If the assembler finds an open parenthesis (character string) with no closing parenthesis (which would make it an embedded-argument), the following warning-level message is issued:

```
Embedded argument was not found.
```

A *syntactically valid expression* can include a legal terminator (space, tab, or semicolon for new format) or an element separator (comma). The syntactically valid expression satisfies the requirements for an expression, but it is used only as an argument and is not evaluated in the macro call itself. If the syntactically valid expression is an embedded argument, then, as long as an asterisk precedes the *positional-parameter* name, the embedded argument is used in its entirety. If an asterisk does not precede the *positional-parameter* name, the outermost parentheses are stripped from the embedded argument and the remaining string

is used as the argument. Use of the syntactically valid expression permits you to enter a string (=' 'R) of characters that may contain one or more spaces or a comma.

The *null string* is an empty string.

- *positional-parameters*

positional-parameters must be specified with valid and unique names and they must meet the requirements for names as described in Section 6.3, page 76. There can be none, one, or more positional parameters. The default argument for a *positional-parameter* is an empty string.

The positional parameters defined in the macro definition are case-sensitive. Positional parameters that are specified in the definition body must identically match positional parameters defined by the macro prototype statement.

The syntax for *keyword-parameters* can be any of the following:

```
[*]name=[expression-argument-value]

[,[keyword-parameters]]

[*] | name=[expression-argument-value]

[,[keyword-parameters]]

[*]name=[string-argument-value]

[,[keyword-parameters]]
```

The elements of *keyword-parameters* are described as follows:

- *keyword-parameters*

keyword-parameters must be specified with valid and unique names. Names within *keyword-parameter* must meet the requirements for names as described in Section 6.3, page 76.

There can be zero, one, or more *keyword-parameters*. You can enter names within *keyword-parameters* in any order. Default arguments can be provided for each *keyword-parameter* at definition time, and they are used if the keyword is not specified at call time.

The *keyword-parameters* defined in a macro definition are case-sensitive. The *keyword-parameters* specified in the macro body must match the *positional-parameters* specified in the macro prototype statement.

The | is optional. If the | is not included, the -s option argument can be an embedded argument, a character string, or a null string. If the | is included, the parameter be either a syntactically valid expression or a null string.

Embedded argument. A left parenthesis signals the beginning of an embedded argument and it must be matched by a right parenthesis. An embedded argument can also contain pairs of matching left and right parentheses. If an asterisk precedes the positional parameter name, the embedded argument is used in its entirety; otherwise, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

Character string. Any character up to but not including a legal terminator (space, tab, or semicolon for new format) or an element separator. If the assembler finds an open parenthesis (character string) with no closing parenthesis (which would make it an embedded argument), the following warning-level listing message is issued:

```
Embedded argument was not found.
```

The null argument is an empty string.

Syntactically valid expression. An expression can include a legal terminator (space, tab, or semicolon for new format) or an element separator (comma). The syntactically valid expression is a legal expression, but it is used only as an argument and is not evaluated in the macro call itself.

If the syntactically valid expression is an embedded argument and if an asterisk precedes the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not precede the *positional-parameter* name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

If a default is provided for a *keyword-parameter*, it must meet the preceding requirements.

The following example illustrates the use of positional parameters:

```
MACRO
JUSTIFY      |PARAM
.                    ; Macro prototype.
.
.
```

```
JUSTIFY  ENDM
         JUSTIFY       ','R ; Macro call
         JUSTIFY       '  'R; Macro call
```

When the following macro is called, the positional parameter p1 receives a value of v1 because an asterisk does not precede the parameter on the prototype statement. The positional parameter p2, however, receives a value of (v2) because an asterisk precedes the parameter on the prototype statement.

```
         macro
         paren  p1,p2      ; Macro prototype.
         .
         .
         .
paren    endm
         paren  (v1),(v2)  ; Macro call.
```

9.2.2 Macro Calls

An instruction of the following format can call a macro definition:

[*locarg*] *functional positional-arguments* ["," [*keyword-arguments*]]

[*locarg*] *functional keyword-arguments*

The elements of the macro call are described as follows:

- *locarg*

  The *locarg* element specifies an optional label field argument. *locarg* must be terminated by a space or a tab (new format only). *locarg* can be any character up to but not including a space. If a label field parameter is specified on the macro definition, you can specify a matching label field parameter on the macro call. *locarg* is substituted wherever the label field parameter occurs in the definition. If no label field parameter is specified in the definition, this field must be empty.

- *functional*

  The *functional* element specifies the macro name. It must be an identifier or an equal sign. *functional* must match the name specified in the macro definition.

- *positional-arguments*

Positional-arguments specify an actual argument string that corresponds to a *positional-parameter* that is specified in the definition prototype statement. The requirements for *positional-arguments* are specified by the corresponding *positional-parameter* in the macro definition prototype statement. *Positional-arguments* are not case-sensitive to *positional-parameters* on the macro call.

The first *positional-argument* is substituted for the first *positional-parameter* in the prototype operand field, the second *positional-argument* string is substituted for the second *positional-parameter* in the prototype operand field, and so on. If the number of *positional-arguments* is less than the number of *positional-parameters* in the prototype operand field, null argument strings are used for the missing *positional-arguments*.

Two consecutive commas indicate a null (empty) *positional-argument* string.

- *keyword-arguments*

  keyword-arguments are an actual argument string that corresponds to a *keyword-parameter* specified in the macro definition prototype statement. The requirements for *keyword-arguments* are specified by the corresponding *keyword-parameter* in the macro definition prototype statement.

  keyword-arguments are not recognized until after *n* subfields (*n* commas); *n* is the number of positional parameters in the operand field of the macro definition.

  You can list *keyword-arguments* in any order; matching the order in which *keyword-parameters* are listed on the macro prototype statement is unnecessary. However, because the *keyword-parameter* is case-sensitive, it must be specified in the macro call exactly as specified in the macro prototype statement to be recognized.

  The default *keyword-parameters* specified in the macro prototype statement are used as the actual *keyword-arguments* for missing *keyword-arguments*.

All arguments must meet the requirements of the corresponding parameters as specified in the macro definition prototype statement.

Note: The | and * are not permitted on the macro call statement. These characters specified in the prototype statement for *positional-parameters* or *keyword-parameters* are remembered by the assembler when the macro is called.

To call a macro, use its name in a code sequence. The INTCONV macro is called as follows:

```
        MACRO
```

```
        INTCONV         P1,P2 ; P1=A reg, P2=Sreg
        P2              +F_P1 ; Transfer with special expression
                              ; and sign extension.
        P2              +F_P2 ; Normalize the S register.
INTCO   ENDM                  ; End of macro definition.
        LIST            MAC
```

Call and expansion of the INTCONV macro:

```
        INTCONV         A1,S3 ; Macro call.
        S2              +FA1  ; Transfer with special expression
                              ; and sign extension.
        S2              +FS2  ; Normalize the S register.
```

Note: Comments preceded by an underscore and an asterisk are included in the definition bodies of the following macro examples. These comments are included to illustrate the way in which parameters are passed from the macro call to the macro definition. Because comments are not assembled, _* comments allow arguments to be shown without regard to hardware differences or available machine instructions.

The following examples show the use of *positional-parameters* and *keyword-parameters.*

The macro table contains positional and keyword parameters.

```
        macro
        table           tabn,val1=#0,val2=,val3=0
tables  section         data
tabn    con             'tabn'1
        con             val1
        con             val2
        con             val3
        section         *   : Resume use of previous section.
table   endm
        list            mac
```

The following shows the call and expansion of the table macro:

```
        table           taba,val3=4,val2=a    ; Macro call.
tables  section         data
taba    con             'taba'1
        con             -
        con             a
        con             4
```

```
        section         *  : Resume use of previous section.
```

Macro noorder demonstrates that *keyword-parameters* are not order dependent.

```
        macro
        noorder         param1,param2,param3=,param4=b
        s1              param1
        s2              param2
        s3              param3
        s4              param4
        noorder endm
        list            mac
```

The call and expansion of the noorder macro is as follows:

```
        noorder         (1),2,param4=dog,param3=d
        s1              1
        s2              2
        s3              d
        s4              dog
```

Macros ONE, two, and THREE demonstrate that the number of parameters specified in the macro call may form the number of parameters specified in the macro definition.

```
        MACRO
        ONE             PARAM1,PARAM2,PARAM3
_*PARAMETER1:           PARAM1
                             ; SYM1 corresponds to PARAM1.
_*PARAMETER2:           PARAM2
                             ; Null string.
_*PARAMETER3:           PARAM3
                             ; Null string.
ONE     ENDM
        LIST            MAC
```

The call and expansion of the ONE macro using one parameter is as follows:

```
        ONE             SYM1 ; Call using one parameter.
* PARAMETER 1:          SYM1 ; SYM1 corresponds to PARAM1.
* PARAMETER 2:               ; Null string.
* PARAMETER 3:               ; Null string.
        macro
        two             param1,param2,param3
_* Parameter 1:         param1
```

```
                              ; SYM1 corresponds to param1.
_* Parameter 2:         param2
                              ; SYM2 corresponds to param2.
_* Parameter 3:         param3
                              ; Null string.
two     endm
        list            mac
```

The call and expansion of the two macro using two parameters is as follows:

```
          two           sym1,sym2 ; Call using two parameters.
* Parameter 1:          sym1 ; sym1 corresponds to param1.
* Parameter 2:          sym2 ; sym2 corresponds to param2.
* Parameter 3:               ; Null string.
        MACRO
        THREE           PARAM1,PARAM2,PARAM3
_*PARAMETER 1:          PARAM1
                              ;SYM1 corresponds to PARAM1.
_*PARAMETER 2:          PARAM2
                              ;SYM2 corresponds to PARAM2.
_*PARAMETER 3:          PARAM3
                              ;SYM3 corresponds to PARAM3.
THREE   ENDM
        LIST            MAC
```

The call and expansion of the THREE macro using prototype parameters is as follows:

```
        THREE           SYM1,SYM2,SYM3 ; Call matching prototype.
* PARAMETER 1:          SYM1 ; SYM1 corresponds to PARAM1.
* PARAMETER 2:          SYM2 ; SYM2 corresponds to PARAM2.
* PARAMETER 3:          SYM3 ; SYM3 corresponds to PARAM3.
```

The following examples demonstrate the use of the optional |.

Macro BANG demonstrates the use of the embedded argument (1,2), syntactically valid expressions for *positional-parameters* ('abc,def'), *keyword-parameters* (PARAM3=1+2), and the null string.

```
        MACRO
        BANG            PARAM1,|PARAM2,|PARAM3=,PARAM4=
_* PARAMETER 1:         PARAM1
                              ; Embedded argument.
_* PARAMETER 2:         PARAM2
                              ; Syntactically valid expression
```

```
_* PARAMETER 3:        PARAM3
                              ; Syntactically valid expression
_* PARAMETER 4:        PARAM4
                              ; Null string.
BANG   ENDM
       LIST            MAC
```

The call and expansion of the BANG macro is as follows:

```
       BANG                (1,2),'abc,def',PARAM3=1+2
                              ; Macro call.
* PARAMETER 1:             1,2
                              ; Embedded argument.
* PARAMETER 2:             'abc,def'
                           ; Syntactically valid expression.
* PARAMETER 3:             1+2
                           ; Syntactically valid expression.
* PARAMETER 4:             ; Null string.
```

In the previous example:

- If the argument for PARAM1 had been (((1,2))), S1 would have received ((1,2)) at expansion.
- The | specified on PARAM2 and PARAM3 permits commas and spaces to be embedded within strings 'abc,def' and allows expressions to be expanded without evaluation 1+2.
- PARAM4 passes a null string. A space or comma following the equal sign specifies a null or empty character string as the default argument.

In the following macro, called remem, the | is remembered from the macro definition when it is called:

```
       macro
       remem  |param1=' 'r  ; Prototype statement includes |
       s1   param1
remem  endm
       list  mac
```

The call and expansion of the remem macro is as follows:

```
remem  param1=',r          ; Macro call does not include |
       s1     ',r
```

The NULL and nullparm macros that follow demonstrate the effect of null strings when parameters are passed.

NULL demonstrates the effect of a null string on macro expansions. P2 is passed a null string. When NULL is expanded, the resulting line is left-shifted two spaces, which is the difference between the length of the parameter (P2) and the null string.

```
        MACRO
        NULL  P1,P2,P3
        S1    P1
        S2    P2                ; Left shifted two places.
        S3    P3
NULL    ENDM
        LIST  MAC
```

The call and expansion of the NULL macro is as follows:

```
        NULL  1,,3              ; Macro call.
        S1    1
*       S2                      ; Left shifted two places.
        S3    3
```

Macro nullparm demonstrates how a macro is expanded when the macro call does not include the label field name specified on the macro definition.

```
        macro
        nullparm      longparm
                              ; Prototype statement.
longparm =              1
nullparm endm
        list           mac
```

The call and expansion of the nulparm macro is as follows:

```
         nullparm
=       1
```

Note: The label field parameter was omitted on the macro call in the previous example. The result and operand fields of the first line of the expansion were shifted left 8 character positions because a null argument was substituted for the 8-character parameter, LONGPARM.

If the old format is used, only one space appears between the label field parameter and result field in the macro definition. If a null argument is substituted for the location parameter, the result field is shifted into the label field in column 2. Therefore, at least two spaces should always appear between a parameter in the label field and the first character in the result field in a definition. If the new format is used, the result field is never shifted into the label field.

The following macro, DEFAULT, illustrates how defaults are assigned for keywords when the macro is expanded:

```
         MACRO
        DEFAULT       PARAM1=(ABC DEF,GHI),PARAM2=ABC,PARAM3=
_* PARAM 1
_* PARAM 2
_* PARAM 3
DEFAULT  ENDM
         LIST         MAC
```

The following illustrates calls and expansions of the DEFAULT macro:

```
        DEFAULT       PARAM1=ARG1,PARAM2=ARG2,PARAM3=ARG3
                           ; Macro call.
*ARG1
*ARG2
*ARG3
        DEFAULT       PARAM1=,PARAM2=(ARG2),PARAM3=ARG3
*ARG2
*ARG3
        DEFAULT       PARAM1=((ARG1)),PARAM2=,PARAM3=ARG3
                           ; Macro call.
* (*ARG1)
* ARG3
```

The following examples illustrate the correct and incorrect way to specify a literal string in a macro definition.

Macro WRONG shows the incorrect way to specify a literal string in a macro definition. The comments in the expansion are writer comments and are not part of the expansion.

```
       MACRO
       WRONG  PARAM1=' 'R    ; Prototype statement.
_* PARAM1
WRONG  ENDM                  ; End of macro definition.
       LIST  MAC             ; List expansion.
```

The call and expansion of WRONG is as follows (the assembler erroneously expands WRONG; ' 'R was intended):

```
       WRONG                 ; Macro call
* '
```

Macro right shows the correct way to specify a literal string in a macro definition.

```
       macro
       right  |param1=' 'r   ; Prototype statement.
* param1
right  endm                  ; End of macro definition.
       list  mac             ; List expansion.
```

The expansion of right is as follows (the assembler expands right as intended because of the |):

```
      right                  ; Macro call.
* ' 'r
```

The following macros demonstrate the wrong and right methods for replacing parameters on the prototype statement with parameters on the macro call statement.

Macro BAD demonstrates the wrong method of replacing parameters.

```
      MACRO
      BAD    PARAM1,PARAM2,PARAM3=JJJ
* PARAMETER 1:        PARAM1
* PARAMETER 2:        PARAM2
* PARAMETER 3:        PARAM3
BAD   ENDM                   ; End of macro definition.
      LIST  MAC              ; Listing expansion.
```

The call and expansion of the BAD macro is as follows:

```
       BAD    PARAM3=XKK      ; Macro call.
* PARAMETER 1:          PARAM3=KKK
* PARAMETER 2:
```

```
* PARAMETER 3:          JJJ
```

Macro good demonstrates the correct method for replacing parameters.

```
        macro
        good  param1,param2,param3=jjj
_* parameter 1:          _param1
                             ; Null string.
_* parameter 2:          param2
                             ; Null string.
_* parameter 3:          param3
good  endm                   ; End of macro definition.
      list  mac              ; Listing expansion.
```

The call and expansion of the good macro is as follows:

```
      good    ,param3=kkk    ; Macro call.
* parameter 1:               ; Null string.
* parameter 2:               ; Null string.
* parameter 3:          kkk
```

Macro ALPHA demonstrates the specification of an embedded parameter.

```
        MACRO                ; EDIT=ON
        ALPHA  |PARAM        ; Appending a string.
_* FORMAL PARM:         PARAM
_* EMBEDDED PARM:       ABC_PARAM_DEFG
                             ; Concatenation off at call time.
ALPHA  ENDM                  ; End of macro definition.
       LIST  MAC             ; List expansion.
```

The call and expansion of the ALPHA macro is as follows:

```
      ALPHA  1               ; Macro call.
* FORMAL PARM:          1
* EMBEDDED PARM:        ABC1DEFG
```

The assembler processes the embedded parameter in macro ALPHA, as follows:

1. The assembler scans the string to identify the parameter. ABC_ cannot be a parameter because the underscore character is not defined as an identifier character for a parameter.
2. The assembler identifies PARAM as the parameter when the second underscore character is encountered.

3. 1 is substituted for PARAM, producing string ABC_1_DEFG.
4. If editing is enabled, the underscore characters are removed and the resulting string is ABC1DEFG.

   If editing is disabled, the string is ABC_1_DEFG.
5. The assembler processes the statement.

9.3 Operation Definitions (OPDEF)

An operation definition (OPDEF) identifies a sequence of statements to be called later in the source program by an opdef call. Each time the opdef call occurs, the definition sequence is placed into the source program.

Opdefs resemble machine instructions and can be used to define new machine instructions or to redefine current machine instructions. Machine instructions map into opcodes that represent some hardware operation. When an operation is required that is not available through the hardware, an opdef can be written to perform that operation. When the opdef is called, the opdef maps into the opdef definition body and the operation is performed by the defined sequence specified in the definition body.

You can replace any existing machine instruction with an opdef. Although opdef definitions should conform to meaningful operations that are supported by the hardware, they are not restricted to such operations.

The opdef definition sets up the parameters into which the arguments specified in the opdef call are substituted. Opdef parameters are always expressed in terms of registers or expressions. The opdef call passes arguments to the parameters in the opdef definition. The syntax for the opdef definition and the opdef call are identical with two exceptions:

- The complex register has been redefined for the opdef definition prototype statement as follows:

```
register_mnemonic.register_parameter
```

- Expressions have been redefined for the opdef definition prototype statement, as follows:

```
@[expression_parameter]
```

These two exceptions allow you to specify parameters in the place of registers and expressions for an opdef definition.

The syntax defining a *register_parameter* and an *expression_parameter* is case-sensitive. Every character that identifies the parameter in the opdef prototype statement must be identical to every character in the body of the opdef definition. This includes the case (uppercase, lowercase, or mixed case) of each character.

Because the opdef can accept arguments in many forms, it can be more flexible than a macro. Opdefs place a greater responsibility for parsing arguments on the assembler. When a macro is specified, the responsibility for parsing arguments is placed on the user in many cases. Parsing a macro argument can involve numerous micro substitutions, which greatly increase the number of statements required to perform a similar operation with an opdef.

Defined sequences (macros, opdefs, dups, and echos) are costly in terms of assembler efficiency. As the number of statements in a defined sequence increases, the speed of the assembler decreases. This decrease in speed is directly related to the number of statements expanded and the number of times a defined sequence is called.

Limiting the number of statements in a defined sequence improves the performance of the assembler. In some cases, an opdef can perform the same operation as a macro and use fewer statements in the process.

The following example illustrates that an opdef can accept many different kinds of arguments from the opdef call:

```
MANYCALL   OPDEF
      A.REG1   A.REG2|A.REG3
                              ; Opdef prototype statement.
      S1   A.REG2
      S2   A.REG3
      A.REG1   S3             ; OR of registers S1 and S2.
MANYCALL   ENDM               ; End of opdef definition.
```

The following example illustrates the calls and expansions of the previous example:

```
      A1    A2|A3             ; First call to opdef MANYCALL.
      S1    A.2
      S2    A.3
      S3    S1|S2
      A.1   S3                ; OR of registers S1 and S2.
      A.1   A.2|A.3           ; Second call to opdef MANYCALL.
```

```
        S1      A.2
        S2      A.3
        S3      S1|S2
        A.1     S3              ; OR of registers S1 and S2.
ONE     =       1               ; Define symbols.
TWO     =       2
THREE   =       3
        A.ONE   A.TWO|A.THREE   ; Third call to opdef MANYCALL.
        S1      A.2
        S2      A.3
        S3      S1|S2
        A.ONE   S3              ; OR of registers S1 and S2.
        A1      A.2|A.THREE     ; Fourth call to opdef MANYCALL.
        S1      A.2
        S2      A.3
        S3      S1|S2
        A.1     s3              ; OR of registers S1 and S2.
```

In the first and second calls to opdef MANYCALL, the arguments passed to REG1, REG2, and REG3 are 1, 2, and 3, respectively. In the third call to opdef MANYCALL, the arguments passed to REG1, REG2, and REG3 are ONE, TWO, and THREE, respectively. The fourth call to opdef MANYCALL demonstrates that the form of the arguments can vary within one call to an opdef if they take a valid form. The arguments passed to REG1, REG2, and REG3 in the fourth call are 1, 2, and THREE, respectively.

The following example illustrates how to use an opdef to limit the number of statements required in a defined sequence:

```
        MACRO
        $IF     REG1,COND,REG2 ; Macro prototype statement.
        .
        .
        .
$IF     ENDM                   ; End of macro definition.
        .
        .
        .
        $IF     S6,EQ,S.3      ; Macro call.
        .
        .
        .
        $ELSE
        .
```

```
.
.
$ENDIF
```

Parsing the parameters (S6,EQ,S3) passed to the definition requires many micro substitutions within the definition body. These micros increase the number of statements within the definition body.

The same function is performed in the following example, but an opdef is specified instead of a macro. In this instance, specifying an opdef rather than a macro reduces the number of statements required for the function.

Because an opdef is called by its form, it is more flexible than a macro in accepting arguments. The opdef expects to be passed two S registers and the EQ mnemonic. You can specify the arguments for the registers in a number of ways and still be recognized as S register arguments by the opdef.

```
         opdef
example  $if  s.reg1,eq,s.reg2 ; Opdef definition statement.
_* Register1: reg1
_* Register2: reg2
example  endm                  ; End of opdef definition.
         list mac              ; Listing expansion.
```

The following are the calls and expansions of the preceding example:

```
         $if              s6,eq,s.3
* Register1:  6
* Register2:  3
```

If an opdef occurs within the global definitions part of a program segment, it is defined as global. Opdef definitions are local if they occur within a program module (an IDENT, END sequence). A global definition can be redefined locally, but the global definition is re-enabled and the local definition is discarded at the end of the program module. You can reference a global definition anywhere within an assembler program after it has been defined.

You can specify the OPDEF pseudo instruction anywhere within a program segment. If the OPDEF pseudo instruction is found within a definition, it is defined. If the OPDEF pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

In the following example, the operand and comment fields of the expanded line are shifted two positions to the left (difference between reg and 1):

```
example  opdef
```

```
        s.reg       @exp  ; Prototype statement.
        a.reg       @exp  ; New machine instruction.
example endm              ; End of opdef definition.
        list        mac   ; Listing expansion.
```

The following are the calls and expansions of the preceding example:

```
s1          2     ; Opdef call.
a.1         2     ; New machine instruction.
```

9.3.1 Opdef Definition

The OPDEF pseudo instruction is the first statement of an opdef definition. Although an opdef is constructed much like a macro, an opdef is defined by an opdef statement, not by a name.

Opdef syntax is uniquely defined on the result field alone, in which case, the operand field is not specified or on the result and operand fields. The OPDEF prototype permits up to five subfields within the result and operand fields. At least one subfield must be present within the result field. No subfields are required in the operand field.

The syntax for each of the subfields within the result and operand fields of the opdef prototype statement is identical. No special syntax forms exist for any of the subfields. The rules that apply for the first subfield in the result field apply to the remainder of the subfields within the result field and to all subfields within the operand field.

The format of the opdef definition is as follows:

name	OPDEF	
[*loc*]	*defsynres*	*defsynop*
	LOCAL	[*name*][, [*name*]]
.		
.		
.		
name	ENDM	

The variables in the opdef definition are described as follows:

- *name*

  name identifies the opdef definition and has no association with functionals that appear in the result field of instructions. *name* must match the name in the label field of the ENDM pseudo instruction, which ends the definition.

- *loc*

  loc specifies an optional label field parameter. *loc* must meet the requirements for names as described in Section 6.3, page 76.

- *defsynres*

  defsynres specifies the definition syntax for the result field. It can be one to five subfields specifying a valid result field syntax. The result field must be a symbolic.

  Valid result subfields for opdefs can be one of the following:

  - Initial register
  - Mnemonic
  - Initial expression

    To specify an initial register on the opdef prototype statement, use one of the following syntax forms for *initial-registers*:

[*prefix*][*register-prefix*] *register*[*register-expression-separatorregister-ending*][*suffix*]
[*prefix*][*register-prefix*] *register*[*register-expression-separatorexpression-ending*][*suffix*]

Note: The formal grammar for the Cray X1 assembly language has changed fairly significantly from the Cray PVP grammar. This section has been partially modified to reflect that, but more changes will be made. The intent is that the grammar reflected in the Cray X1 instruction set be generalized for opdefs. A formal grammar will be presented in the near future.

The elements of an initial register definition are as follows:

- *prefix*

  prefix is optional and can be either a left parenthesis or a left bracket ([).

- *register-prefix*

register-prefix is optional. It can be specified as any of the following characters:
`+ - ~ !`

- *register*

  register is required. It can be any simple or complex register. The only simple register is VL.

The complex registers are designated in the opdef definition in the form: *register_designator.register_parameter*. The *register_designator* for complex registers can be any of the following: `A C M S T`.

The *register-parameter* is a 1- to 8-character identifier composed of identifier characters.

When you specify a *simple register* or a *complex register* mnemonic on an opdef call, it is recognized by the opdef definition without regard to the case (uppercase, lowercase, or mixed case) in which it was entered.

The optional *register-expression-separator* can be designated by any of the following:

`& | ^ ~ << >> + - * / ? : = <= >=`

- *register-ending*

The optional *register-ending* is specified using the following syntax form:

register [*register-expression-separator register-or-expression*]

The *register* and *register-expression-separator* elements are described previously under *initial-register*.

The optional *register-or-expression* can be a register or an expression. If *register* is not specified, *expression* is required. If *expression* is not specified, *register* is required.

expression has been redefined for the opdef prototype statement, as *expression-parameter*. *expression-parameter* is an identifier that must begin with the at symbol (@). The @ can be followed by 0 to 7 identifier characters.

- *expression-ending*

expression-ending is specified as follows:

expression [*register_*
expression_separator [*register-or-expression*]

expression is required and has been redefined for the opdef prototype statement as follows:

expression-parameter

expression-parameter is an identifier that must begin with the at symbol (@). The @ can be followed by 0 to 7 identifier characters.

register_expression-separator is defined above.

The optional *register-or-expression* can be a register or an expression. If *register* is not specified, *expression* is required. If *expression* is not specified, *register* is required.

A mnemonic is a 1- to 8-character identifier that must begin with a letter (A through Z or a through z), a decimal digit (0 through 9), or one of the following characters: $, %, &, ', *, +, -, ., /, :, =, ?,, `, |, or ~. Optional characters 2 through 8 can be the at symbol (@) or any of the previously mentioned characters. Mnemonics are case-insensitive.

Initial-expression specifies an *initial-expression* on the opdef prototype statement, use one of the following syntax forms for *initial-expressions*:

prefixl [*expression-prefix*] *expression* [*expression-separator* [*register-ending*]]

[*prefix*] [*expression-prefix*] *expression* [*expression-separator* [*expression-ending*]]

expression [*expression-separator* [*register-ending*]]

expression [*expression-separator* [*expression-ending*]]

The elements of the initial expression are described as follows:

- *prefix*

prefix is optional and can be either a right parenthesis or a right bracket (]).

- *expression-prefix*

expression-prefix is optional and can be any of the following:

`<, >, #<, or #>`

- *expression*

expression is required and has been redefined for the opdef prototype statement, as follows:

expression-parameter

expression-parameter is an identifier that must begin with the at symbol (@). The @ can be followed from 0 to 7 identifier characters.

- *expression-separator*

expression-separator is optional and can be one of the following:

`), ], &, |, \, <, >, #<, or #>`

- *register-ending* and *expression-ending*

register-ending and *expression-ending* are the same for initial expressions as for initial registers.

- *defsynop*

  Definition syntax for the operand field; can be zero, one, or two subfields specifying a valid operand field syntax. If a subfield exists in the result field, the first subfield in the operand field must be a symbolic.

  The definition syntax for the operand field of an opdef is the same as the definition syntax for the result field of an opdef. See the definition of *defsynres*, earlier in this subsection.

9.3.2 Opdef Calls

An opdef definition is called by an instruction that matches the syntax of the result and operand fields as specified in the opdef prototype statement.

The arguments on the opdef call are passed to the parameters on the opdef prototype statement. The special syntax for registers and expressions that was required on the opdef definition does not extend to the opdef call.

The format of the opdef call is as follows:

locarg	*callsynres*	*callsynop*

The variables associated with the opdef call are described as follows:

- *locarg*

  locarg is an optional label field argument. It can consist of any characters and is terminated by a space (embedded spaces are illegal).

  If a label field parameter is specified on the opdef definition, a matching label field parameter can be specified on the opdef call. *locarg* is substituted wherever the label field parameter occurs in the definition. If no label field parameter is specified in the definition, this field must be empty.

- *callsynres*

  callsynres specifies the result field syntax for the opdef call. It can consist of one, two, or three subfields and must have the same syntax as specified in the result field of the opdef definition prototype statement.

  The syntax of the result field call is the same as the syntax of the result field definition with two exceptions. The special syntax rules that are in effect for registers and expressions on the opdef definition do not apply to the opdef call. The syntax for registers and expressions used on the opdef call is the same as the syntax for registers and expressions.

  The subfields in the result field on the opdef call can be specified with one of the following:

  - Initial-register
  - Mnemonic
  - Initial-expression

    For a description of the syntax for the result field of the opdef call, see the syntax for the result field of the opdef definition.

- *callsynop*

  callsynop specifies the operand field syntax for the opdef call. It can consist of zero, one, two, or three subfields, and it must have the same syntax as specified in the operand field of the opdef definition prototype statement.

  The syntax of the operand field call is the same as the syntax of the operand field definition with two exceptions. The special syntax rules that are in effect for registers and expressions on the opdef definition do not apply to the opdef call. The syntax for registers and expressions used on the opdef call is the same as the syntax for registers and expressions.

  The subfields in the operand field on the opdef call can be specified with one of the following:

- Initial-register
- Mnemonic
- Initial-expression

  For a description of the syntax for the operand field of the opdef call, see the syntax for the result field of the opdef definition.

The following rules apply for opdef calls:

- The character strings *callsynres* and *callsynop* must be exactly as specified in the opdef definition.
- An expression must appear whenever an expression in the form @ *exp* is indicated in the prototype statement. The actual argument string is substituted in the definition sequence wherever the corresponding formal parameter @ *exp* occurs.
- The actual argument string consisting of a *complex-register* mnemonic followed by a period (.) followed by a *register-parameter*. A *register-designator* followed by a *register-parameter* must appear wherever the *register-designator* `A.` *register-parameter*, `B.` *register-parameter*, `SB.` *register-parameter*, `S.` *register-parameter*, `T.` *register-parameter*, `ST.` *register-parameter*, `SM.` *register-parameter*, or `V.` *register-parameter*, respectively, appeared in the prototype statement.
  - If the *register-parameter* is of the form *octal-integer*, the actual argument is the *octal-integer* part. The *octal-integer* is restricted to 4 octal digits.
  - If the *register-parameter* is of the form . *integer-constant* or . *symbol*, the actual argument is an *integer-constant* or a symbol.

The following opdef definition shows a scalar floating-point divide sequence:

```
fdv  opdef                          ; Scalar floating-point divide
                                    ; prototype statement.
L    s.r1      s.r2/fs.r3
     errif     r1,eq,r2
     errif     r1,eq,r3
L    s.r1      /hs.r3
     s.r2      s.r2*fs.r1
     s.r3      s.r3*is.r1
     s.r1      s.r2*fs.r3
fdv  endm
```

The following example illustrates the opdef call and expansion of the preceding example:

```
a     s4     s3/fs2           ; Divide s3 by s2, result to s4.
      errif  4,eq,3
      errif  4,eq,2
a     s.4    /hs.2
      s.3    s.3*fs.4
      s.2    s.2*is.4
      s.4    s.3*fs.2
```

The following opdef definition, call, and expansion define a conditional jump where a jump occurs if the A register values are equal:

```
JEQ  OPDEF
L    JEQ    A.A1,A.A2,@TAG  ; Opdef prototype statement.
L    A0     A_A1-A_A2
_*   JAZ    @TAG            ; Expression is expected.
JEQ  ENDM                   ; End of opdef definition.
     LIST   MAC              ; Listing expansion.
```

The following example illustrates the opdef call and expansion of the preceding example (the expansion starts on line 2.):

```
      JEQ   A3,A6,GO           ; Opdef call.
      A0    A3-A5
*     JAZ   GO                 ; Expression is expected.
```

The opdef in the following example illustrates how an opdef can redefine an existing machine instruction:

```
EXAMPLE  OPDEF
         S.REG            @EXP ; Opdef protype instruction.
         A.REG            @EXP ; New instruction.
EXAMPLE  ENDM                  ; End of opdef definition.
         LIST  MAC             ; Listing expansion.
```

The following example illustrates the opdef call and expansion of the preceding example:

```
      S1  2          ; Opdef call.
      A.1  2         ; New instruction.
```

The following example demonstrates how the expansion of an opdef is affected when the opdef call does not include a label that was specified in the opdef definition:

```
regchg opdef
lbl   s.reg1 s.reg2          ; Opdef prototype statement.
lbl   =    *                 ; Left-shift if lbl is left off.
      s.reg2 s.reg1          ; Register s2 gets register s1.
regchg endm                  ; End of opdef definition.
       list  mac             ; Listing expansion.
```

The following example illustrates the opdef call and expansion of the preceding example:

```
          s1  s2         ; Opdef call.
      =   *              ; Left-shift if lbl is left off.
          s.2  s.1       ; Register s2 gets register s1.
```

The label field parameter was omitted on the opdef call in the previous example. The result and operand fields of the first line of the expansion were shifted left three character positions because a null argument was substituted for the 3-character parameter, `lbl`.

If the old format is used, only one space appears between the label field parameter and result field in the macro definition. If a null argument is substituted for the location parameter, the result field is shifted into the label field in column 2. Therefore, at least two spaces should always appear between a parameter in the label field and the first character in the result field in a definition.

If the new format is used, the result field is never shifted into the label field.

The following example illustrates the case insensitivity of the register and register-prefix:

```
CASE   OPDEF
       S1     #Pa2          ; Prototype statement.
       .
       .
       .
CASE   ENDM
```

The following example illustrates the opdef calls of the preceding example:

```
   S1  #pa2              ; Recognized by CASE.
   S1  #Pa2              ; Recognized by CASE.
```

```
S1  #pA2            ; Recognized by CASE.
S1  #PA2            ; Recognized by CASE.
s1  #pa2            ; Recognized by CASE.
s1  #Pa2            ; Recognized by CASE.
s1  #pA2            ; Recognized by CASE.
s1  #PA2            ; Recognized by CASE.
```

9.4 Duplication (DUP)

The DUP pseudo instruction defines a sequence of code that is assembled repetitively immediately following the definition. The sequence of code is assembled the number of times specified on the DUP pseudo instruction. The sequence of code to be repeated consists of the statements following the DUP pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The sequence to be duplicated ends when the statement count is exhausted or when an ENDDUP pseudo instruction with a matching label field name is encountered.

The DUP pseudo instruction only accepts one type of formal parameter. That parameter must be specified with the LOCAL pseudo instruction.

You can specify the DUP pseudo instruction anywhere within a program segment. If the DUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the DUP pseudo instruction is as follows:

[*dupname*]	DUP	*expression*[, [*count*]]

The variables associated with the DUP pseudo instruction are described as follows:

- *dupname*

  dupname specifies an optional name for the dup sequence. It is required if the *count* field is null or missing. If no count field is present, *dupname* must match an ENDDUP name. The sequence field in the DUP pseudo instruction itself represents the nested dup level and appears in columns 89 and 90 on the listing. For a description of sequence field nest level numbering, see Section 9.1, page 214.

The *dupname* variable must meet the requirements for names as described in Section 6.3, page 76.

- *expression*

  expression is an absolute expression with a positive value that specifies the number of times to repeat the code sequence. All symbols, if any, must be defined previously. If the current base is mixed, octal is used for the expression. If the value is 0, the code is skipped. You can use a STOPDUP to override the given expression.

  The *expression* operand must meet the requirements for expressions as described in Section 6.9, page 94.

- *count*

  count is an optional absolute expression with positive value that specifies the number of statements to be duplicated. All symbols (if any) must be defined previously. If the current base is mixed, octal is used for the expression.

  LOCAL pseudo instructions and comment statements (* in column 1) are ignored for the purpose of this count. Statements are counted before expansion of nested macro or opdef calls, and dup or echo sequences.

  The *count* operand must meet the requirements for expressions as described in Section 6.9, page 94.

In the following example, the code sequence following the DUP pseudo instruction will be repeated 3 times. There are 5 statements in the sequence.

```
      DUP    3,5
      LOCAL   SYM1,SYM2       ; LOCAL pseudo instruction not counted.
*Asterisk comment; not counted
      S1      1               ; First statement is definition.
*Asterisk comment; not counted
      INCLUDE  ALPHA          ; INCLUDE pseudo instruction not
                              ; counted.
```

The following is the file, ALPHA:

```
    S2   3              ; Second statement in definition.
    S4    4             ; Third statement in definition.
*Asterisk comment        ; not counted
    S5    5             ; Fourth statement in definition.
    S6    6             ; Fifth statement in definition.
```

In the following example, the two con pseudo instructions are duplicated three times immediately following the definition:

```
         list      dup
example  dup       3         ; Definition.
         con       1
         con       2
example  enddup
```

The following example illustrates the expansion of the preceding example:

```
con   1
con   2
con   1
con   2
con   1
con   2
```

9.5 Duplicate with Varying Argument (ECHO)

The ECHO pseudo instruction defines a sequence of code that is assembled zero or more times immediately following the definition. On each repetition, the actual arguments are substituted for the formal parameters until the longest argument list is exhausted. Null strings are substituted for the formal parameters after shorter argument lists are exhausted. The echo sequence to be repeated consists of statements following the ECHO pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The echo sequence ends with an ENDDUP that has a matching label field name.

You can use the STOPDUP pseudo instruction to override the repetition count determined by the number of arguments in the longest argument list.

You can specify the ECHO pseudo instruction anywhere within a program segment. If the ECHO pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ECHO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ECHO pseudo instruction is as follows:

dupname	ECHO	[*name=argument*], [[*name=*]*argument*]

The variables associated with the ECHO pseudo instruction are described as follows:

- *dupname*

  dupname specifies the required name of the echo sequence. It must match the label field name in the ENDDUP instruction that terminates the echo sequence. *dupname* must meet the requirements for names as described in Section 6.3, page 76.

- *name*

  name specifies the formal parameter name. It must be unique. There can be none, one, or more formal parameters. *name* must meet the requirements for names as described in Section 6.3, page 76.

- *argument*

  argument specifies a list of actual arguments. The list can be one argument or a parenthesized list of arguments.

  A single argument is any ASCII character up to but not including the element separator, a space, a tab (new format only), or a semicolon (new format only). The first character cannot be a left parenthesis.

  A parenthesized list can be a list of one or more actual arguments. Each actual argument can be one of the following:

  - An ASCII character string can contain embedded arguments. If, however, an ASCII string is intended, the first character in the string cannot be a left parenthesis. A legal ASCII string is 4(5). An illegal ASCII string is (5)4(5).
  - A null argument; an empty ASCII character string.
  - An embedded argument that contains a list of arguments enclosed in matching parentheses. An embedded argument can contain blanks or commas and matched pairs of parentheses. The outermost parentheses are always stripped from an embedded argument when an echo definition is expanded.

An embedded argument must meet the requirements for embedded arguments as described in page 216.

In the following example, the ECHO pseudo instruction is expanded twice immediately following the definition:

```
        LIST    DUP
EXAMPLE ECHO    PARAM1=(1,3),PARAM2=(2,4)
```

```
                        ; Definition.
          CON       PARAM1
                        ; Gets 1 and 3.
          CON       PARAM2
                        ; Gets 2 and 4.
EXAMPLE   ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
     CON    1              ; Gets 1 and 3.
     CON    2              ; Gets 2 and 4.
     CON    3              ; Gets 1 and 3.
     CON    3              ; Gets 1 and 3.
     CON    4              ; Gets 2 and 4.
```

In the following example, the echo pseudo instruction is expanded once immediately following the definition with two null arguments.

```
          list          dup
example   echo          param1=,param2=()
                            ; ECHO with two null parameters.
_*Parameter 1 is:         'param1'
_*Parameter 2 is:         'param2'
example  enddup
```

The following illustrates the expansion of the preceding example:

```
*Parameter 1 is:      ''
*Parameter 2 is:      ''
```

9.6 Ending a Macro or Operation Definition (ENDM)

An ENDM pseudo instruction terminates the body of a macro or opdef definition. If ENDM is used within a MACRO or OPDEF definition with a different name, it has no effect.

You can specify the ENDM pseudo instruction only within a macro or opdef definition. If the ENDM pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ENDM pseudo instruction is as follows:

```
func      ENDM      ignored
```

The *func* variable associated with the ENDM pseudo instruction identifies the name of the macro or opdef definition sequence. It must be a valid identifier or the equal sign. *func* must match the name that appears in the result field of the macro prototype or the label field name in an OPDEF instruction.

If the ENDM pseudo instruction is encountered within a definition but *func* does not match the name of an opdef or the name of a macro, the ENDM instruction is defined and does not terminate the opdef or macro definition in which it is found. *func* must meet the requirements for functionals.

9.7 Premature Exit from a Macro Expansion (EXITM)

The EXITM pseudo instruction immediately terminates the innermost nested macro or opdef expansion, if any, caused by either a macro or an opdef call. If files were included within this expansion and/or one or more dup or echo expansions are in progress within the innermost macro or opdef expansion they are also terminated immediately. If such an expansion does not exist, the EXITM pseudo instruction issues a caution level listing message and does nothing.

You can specify the EXITM pseudo instruction anywhere within a program segment. If the EXITM pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the EXITM pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the EXITM pseudo instruction is as follows:

```
ignored     EXITM     ignored
```

In the following example of a macro call, the macro expansion is terminated immediately by the EXITM pseudo instruction and the second comment is not included as part of the expansion:

```
      macro
      alpha
_*First comment
      exitm
_*Second comment
alpha  endm
       list  mac
```

The following example illustrates the expansion of the preceding example:

```
        alpha                   ; Macro call
*First  comment
        exitm
```

9.8 Ending Duplicated Code (ENDDUP)

The ENDDUP pseudo instruction ends the definition of the code sequence to be repeated. An ENDDUP pseudo instruction terminates a dup or echo definition with the same name. If ENDDUP is used within a DUP or ECHO definition with a different label field name, it has no effect. ENDDUP has no effect on a dup definition terminated by a statement count; in this case, ENDDUP is counted.

The ENDDUP pseudo instruction is restricted to definitions (DUP or ECHO). If the ENDDUP pseudo instruction is found on a MACRO or OPDEF definition, it is defined and is not recognized as a pseudo instruction. If the ENDDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ENDDUP pseudo instruction is as follows:

dupname `ENDDUP ignored`

The *dupname* variable associated with the ENDDUP pseudo instruction specifies the required name of a dup sequence. *dupname* must meet the requirements for names as described in Section 6.3, page 76.

9.9 Premature Exit of the Current Iteration of Duplication Expansion (NEXTDUP)

The NEXTDUP pseudo instruction stops the current iteration of a duplication sequence indicated by a DUP or an ECHO pseudo instruction. Assembly of the current repetition of the dup sequence is terminated immediately and the next repetition, if any, is begun.

Assembly of the current iteration of the innermost duplication expansion with a matching label field name is terminated immediately. If the label field name is not present, assembly of the current iteration of the innermost duplication expansion is terminated immediately.

If other dup, echo, macro, or opdef expansions were included within the duplication expansion to be terminated, these expansions are also terminated immediately. If a file also is being included at expansion time within the duplication expansion it is terminated immediately.

You can specify the NEXTDUP pseudo instruction anywhere within a program segment. If the NEXTDUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the NEXTDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo.

The format of the NEXTDUP pseudo instruction is as follows:

```
[dupname]      NEXTDUP      ignored
```

The optional *dupname* variable specifies the name of a dup sequence. If the name is present but does not match any existing duplication expansion, a caution-level listing message is issued and the pseudo instruction does nothing. If the name is not present and a duplication expansion does not currently exist, a caution-level listing message is issued and the pseudo instruction does nothing.

9.10 Stopping Duplication (STOPDUP)

The STOPDUP pseudo instruction stops duplication of a code sequence indicated by a DUP or ECHO pseudo instruction. STOPDUP overrides the repetition count.

Assembly of the current dup sequence is terminated immediately. STOPDUP terminates the innermost dup or echo sequence with the same name as found in the label field. If no label field name exists, STOPDUP will terminate the innermost dup or echo sequence. STOPDUP does not affect the definition of the code sequence that will be duplicated.

Assembly of the innermost duplication expansion with a matching label field name is terminated immediately; however, if the label field name is not present, assembly of the innermost duplication expansion is terminated immediately. If other dup, echo, macro, or opdef expansions were included within the duplication expansion that will be terminated, these expansions also are terminated immediately. If a file also is being included at expansion time within the duplication expansion that will be terminated, the inclusion of that file is terminated immediately.

You can specify the STOPDUP pseudo instruction anywhere within a program segment. If the STOPDUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the STOPDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the STOPDUP pseudo instruction is as follows:

```
[dupname]      STOPDUP      ignored
```

The *dupname* variable associated with the STOPDUP pseudo instruction specifies the name of a dup sequence. If the name is present but does not match any existing duplication expansion, or, if the name is not present and a duplication expansion does not currently exist, a caution-level listing message is issued and the pseudo instruction does nothing. *dupname* must meet the requirements for names as described in Section 6.3, page 76.

The following example uses a DUP pseudo instruction to define an array with values 0, 1, and 2:

```
S  =    W*
   DUP  3,1
   CON  W.*-S
```

The following illustrates the expansion of the preceding example:

```
   CON  W.*-S
   CON  W.*-S
   CON  W.*-S
```

In the following example the ECHO and DUP pseudo instructions define a nested duplication:

```
ECHO   ECHO              RI=(A,S),RJK=(B,T)
I      SET               0
DUPI   DUP               8
JK     SET               0
DUPJK  DUP               64
       RI.I              RJK.JK
JK     SET               JK+1
DUPJK  ENDDUP
I      SET               I+1
DUPI   ENDDUP
ECHO   ENDDUP
```

Note: The following expansion is not generated by the assembler, but it is included to show the expansion of the previously nested duplication expansion.

The following example illustrates the expansion of the preceding example:

```
                        ; In the first call of the echo, the A
                        ; and B parameters are used.
    A.0     B.0         ; DUPJK generates the A.0 gets register
    .       .           ; B.0 through register A.0 gets register
    .       .           ; B.64 instructions.
    .       .
    A.0     B.64        ; DUPI increments the A register from
    .       .           ; A.1 to A.7 for succeeding passes
    .       .           ; through DUPJK.
    .       .
    A.1     B.0         ; DUPJK generates register A.i gets
    .       .           ; register B.0 through register A.i gets
    .       .           ; register B.64 instructions.
    .       .
    A.7     B.64        ; i is 1 to 7.
    S.0      T.O        ; In the second expansion of the echo
    .       .           ; pseudo instruction the S and T
    .       .           ; parameters are used.
    .       .
    S.0     T.64        ; DUPJK and DUPI generate the same
    .       .           ; series of register instructions for
    .       .           ; the S and T registers that were
    .       .           ; generated for the A and B registers.
    S.8      T.64
```

In the following example the STOPDUP pseudo instruction terminates duplication:

```
    LIST    DUP
T   SET     0
A   DUP     1000
T   SET     T+1
    IFE     T,EQ,3,1        ; Terminate duplication when T=3.
A   STOPDUP
    CON     T
A   ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
T    SET    T+1
     CON    T
T    SET    T+1
```

```
      CON     T
T     SET     T+1
A     STOPDUP
```

In the following example a STOPDUP pseudo instruction is used to terminate a DUP immediately:

```
DNAME       DUP             3
_* First comment
            STOPDUP
_* Second comment
DNAME       ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
* First comment
          STOPDUP
```

The following example is similar to the previous example except NEXTDUP replaces STOPDUP. The current iteration is terminated immediately when the NEXTDUP pseudo instruction is encountered.

```
DNAME    DUP              3
_* First comment
          NEXTDUP
_* Second comment
DNAME     ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
* First comment
          NEXTDUP
* First comment
          NEXTDUP
* First comment
          NEXTDUP
```

9.11 Specifying Local Unique Character String Replacements (LOCAL)

The LOCAL pseudo instruction specifies unique character string replacements within a program segment that are defined only within the macro, opdef, dup, or echo definition. These character string replacements are known only in the macro, opdef, dup, or echo at expansion time. The most common usage of the LOCAL pseudo instruction is for defining symbols, but the LOCAL

pseudo instruction is not restricted to the definition of symbols. Local pseudo instructions within a macro, opdef, dup, or echo header are not part of the macro definition.

On each macro or opdef call and each repetition of a dup or echo definition sequence, the assembler creates a unique 8-character string (commonly used for the definition of symbols by the user) for each local parameter and substitutes the created string for the local parameter on each occurrence within the definition. The unique character string created for local parameters has the form `%%` *nnnnnn*; where *n* is a decimal digit.

Zero or more LOCAL pseudo instructions can appear in the header of a macro, opdef, dup, or echo definition. The LOCAL pseudo instructions must immediately follow the macro or opdef prototype statement or DUP and ECHO pseudo instructions, except for intervening comment statements.

You can specify the LOCAL pseudo instruction only within a definition. If the LOCAL pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the LOCAL pseudo instruction is as follows:

```
ignored  LOCAL     [name][,[name]]
```

The *name* variable associated with the LOCAL pseudo instruction specifies formal parameters that must be unique and will be rendered local to the definition. *name* must meet the requirements for names as described in Section 6.3, page 76.

The following example demonstrates that all formal parameters must be unique:

```
MACRO
UNIQUE   PARM2          ; PARM2 is defined within UNIQUE.
LOCAL    PARM1,PARM2    ; ERROR: PARM2 previously defined as a
.        .              ; parameter in the macro prototype
.        .              ; statement.
.        .
UNIQUE   ENDM
```

The following example demonstrates how a unique character string is generated for each parameter defined by the LOCAL pseudo instruction:

```
macro
string
local param1,param2  ; Not part of the definition body.
```

```
param1 =    1
            s1   param1      ; Register s1 gets the value defined by
                             ; param1.
param2 =    2
            s2   param2      ; Register s2 gets the value defined by
                             ; param2.
string endm
                             ; End of macro definition.
       list  mac             ; Listing expansion.
```

The following example illustrates the call and expansion from the preceding example:

```
         string              ; Macro call.
%%262144 =          1
         s1         %%262144
                             ; Register s1 gets the value defined by
                             ; param1.
%%131072 =          2
         s2         %%131072
                             ; Register s2 gets the value defined by
                             ; param2.
```

The call to the macro string generates unique strings for `param1` (`%%262144`) and for `param2` (`%%131072`).

9.12 Synonymous Operations (OPSYN)

The OPSYN pseudo instruction defines an operation that is synonymous with another macro or pseudo instruction operation. The name in the label field is defined as being the same as the name in the operand field. You can redefine any pseudo instruction or macro in this manner.

The name in the label field can be a currently defined macro or pseudo instruction in which case, the current definition is replaced and a message is issued informing you that a redefinition has occurred.

An operation defined by OPSYN is global if the OPSYN pseudo instruction occurs within the global part of an assembler segment, and it is local if the OPSYN pseudo instruction appears within an assembler module of a segment. You can reference global operations in any program segment following the definition. Every local operation is removed at the end of a program module, making any previous global definition with the same name available again.

If the OPSYN pseudo instruction occurs within a definition, it is defined and is not recognized as a pseudo instruction. If the OPSYN pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the OPSYN pseudo instruction is as follows:

func1	OPSYN	[*func2*]

The *func1* variable associated with the OPSYN pseudo instruction specifies a required name. It must be a valid name. The name of a defined operation such as a pseudo instruction or macro, or the equal sign. *func* 1 must not be blank and must meet the requirements for names.

The *func2* variable specifies an optional name. It must be the name of a defined operation or the equal sign. If *func* 2 is blank, *func* 1 becomes a do-nothing pseudo instruction.

In the following example, the macro definition includes the OPSYN pseudo instruction that redefines the IDENT pseudo instruction:

```
IDENTT      OPSYN      IDENT
            MLEVEL     ERROR ; Eliminates the warning error that is
                             ; issued because the IDENT pseudo
                             ; instruction is redefined.
        MACRO
        IDENT          NAME
        LIST           LIS,OFF,NXRF
NAME    LIST           LIS,ON,XRF
                             ; Processed if LIST=NAME on CAL control
                             ; statement.
        IDENTT         NAME
IDENT   ENDM
```

The following example illustrates the OPSYN call and expansion (The expansion starts on line 2.):

```
    IDENT   A
    LIST    LIS,OFF,NXRF
A   LIST    LIS,ON,XRF      ; Processed if LIST=NAME on CAL control
                            ; statement.
    IDENTT  A
```

In the following example, the `first` macro illustrates that a functional can be redefined many times:

```
        macro
        first
        s1          1
        s2          2
        s3          s1+2
first   endm
second  opsyn       first
                       ; second is the same as first.
third   opsyn       second
                       ; third is the same as second.
```

The following example includes the opdef calls and expansions from the preceding example:

```
first                ; Macro call.
s1          1
s2          2
s3          s1+s2
second      s1
s1          1
s2          2
s3          s1+s2
third       s1
s1          1
s2          2
s3          s1+s2
```

In the following example, the functional EQU is defined to perform the same operation as =:

```
EQU  OPSYN  =          ; EQU is defined to
                       ; perform the
                       ; operation that the
                       ; = pseudo
                       ; instruction
                       ; performs.
```

ASDEF Macros and Opdefs [A]

(Deferred implementation)

Character Set [B]

Table 19 lists the character sets supported by CAL.

Table 19. Character Set

Character	ASCII code (octal/hex)
NUL	000/00
SOH	001/01
STX	002/02
ETX	003/03
EOT	004/04
ENQ	005/05
ACK	006/06
BEL	007/07
BS	010/08
HT	011/09
LF	012/0A
VT	013/0B
FF	014/0C
CR	015/0D
SO	016/0E
SI	017/0F
DLE	020/10
DC1	021/11
DC2	022/12
DC3	023/13
DC4	024/14
NAK	025/15
SYN	026/16

Character	ASCII code (octal/hex)
ETB	027/17
CAN	030/18
EM	031/19
SUB	032/1A
ESC	033/1B
FS	034/1C
GS	035/1D
RS	036/1E
US	037/1F
Space	040/20
!	041/21
"	042/22
#	043/23
$	044/24
%	045/25
&	046/26
'	047/27
(	050/28
)	051/29
*	052/2A
+	053/2B
,	054/2C
-	055/2D
.	056/2E
/	057/2F
0	060/30
1	061/31
2	062/32

Character	ASCII code (octal/hex)
3	063/33
4	064/34
5	065/35
6	066/36
7	067/37
8	070/38
9	071/39
:	072/3A
;	073/3D
<	074/3C
=	075/3D
>	076/3E
?	077/3F
@	100/40
A	101/41
B	102/42
C	103/43
D	104/44
E	105/45
F	106/46
G	107/47
H	110/48
I	111/49
J	112/4A
K	113/4B
L	114/4C
M	115/4D
N	116/4E

Character	ASCII code (octal/hex)
O	117/4F
P	120/50
Q	121/51
R	122/52
S	123/53
T	124/54
U	125/55
V	126/56
W	127/57
X	130/58
Y	131/59
Z	132/5A
[	133/5B
\	134/5C
]	135/5D
^	136/5E
_	137/5F
`	140/69
a	141/61
b	142/62
c	143/63
d	144/64
e	145/65
f	146/66
g	147/67
h	150/68
i	151/69
j	152/6A

Character	ASCII code (octal/hex)
k	153/6B
l	154/6C
m	155/6D
n	156/6E
o	157/6F
p	160/70
q	161/71
r	162/72
s	163/73
t	164/74
u	165/75
v	166/76
w	167/77
x	170/78
y	171/79
z	172/7A
{	173/7B
}	174/7C
\| | 175/7D |
~ | 176/7E |
DEL | 177/7F |

Glossary

absolute address

1. A unique, explicit identification of a memory location, a peripheral device, or a location within a peripheral device. 2. A precise memory location that is an actual address number rather than an expression from which the address can be calculated.

application node

For Cray X1 systems, a node that is used to run user applications. Application nodes are best suited for executing parallel applications and are managed by the strong application placement scheduling and gang scheduling mechanism Psched. See also *node*; *OS node*; *support node*.

barrier

An obstacle within a program that provides a mechanism for synchronizing tasks. When a task encounters a barrier, it must wait until all specified tasks reach the barrier.

breakpoint

A point in a program that, when reached, triggers some special behavior useful to the process of debugging; generally, breakpoints are used to either pause program execution and/or dump the values of some or all of the program variables. Breakpoints may be part of the program itself, or they may be set by the programmer as part of an interactive session with a debugging tool for scrutinizing the execution of the program.

cache line

A division of cache. Each cache line can hold multiple data items. For Cray X1 systems, a cache line is 32 bytes, which is the maximum size of a hardware message.

CPU

For Cray X1 systems, a multistreaming processor (MSP).

Cray Fortran Compiler

The compiler that translates Fortran programs into Cray object files. The Cray Fortran Compiler fully supports the Fortran language through the Fortran 95 Standard, ISO/IEC 1539-1:1997. Selected features from the proposed Fortran 2003 Standard are also supported.

CrayDoc

Cray's documentation system for accessing and searching Cray books, man pages, and glossary terms in HTML and/or PDF format from a web browser. CrayDoc runs on any operating system based on a UNIX or Linux operating system.

distributed memory

The kind of memory in a parallel processor where each processor has fast access to its own local memory and where to access another processor's memory it must send a message via the interprocessor network.

entry point

A location in a program or routine at which execution begins. A routine may have several entry points, each serving a different purpose. Linkage between program modules is performed when the linkage editor binds the external references of one group of modules to the entry points of another module.

environment variable

A variable that stores a string of characters for use by your shell and the processes that execute under the shell. Some environment variables are predefined by the shell, and others are defined by an application or user. Shell-level environment variables let you specify the search path that the shell uses to locate executable files, the shell prompt, and many other characteristics of the operation of your shell. Most environment variables are described in the ENVIRONMENT VARIABLES section of the man page for the affected command.

gather/scatter

An operation that copies data between remote and local memory or within local memory. A gather is any software operation that copies a set of data that is nonsequential in a remote (or local) processor, usually storing into a sequential (contiguous) area of local processor memory. A scatter copies data from a sequential, contiguous area of local processor memory) into nonsequential locations in a remote (or local) memory.

IRIX

A version of the UNIX System V operating system that is produced by Silicon Graphics, Inc.

loader table

The form in which code is presented to the loader. Loader tables, which are generated by compilers and assemblers according to loader requirements, contain information required for loading. This information includes the type of code; names, types, and lengths of storage blocks; data to be stored; and so on.

Modules

A package on the Cray X1 system that allows you to dynamically modify your user environment by using module files. (This term is not related to the module statement of the Fortran language; it is related to setting up the Cray X1 system environment.) The user interface to this package is the `module` command, which provides a number of capabilities to the user, including loading a module file, unloading a module file, listing which module files are loaded, determining which module files are available, and others.

MSP mode (multistreaming mode)

One of two types of application modes. Programs are compiled either as MSP-mode applications (default) or SSP-mode applications. MSP-mode applications run on one or more MSPs. For MSP-mode applications, each MSP coordinates the interactions of its associated four SSPs. See also command mode; *SSP mode.*

multichip module (MCM)

For Cray X1 systems, the packaging that contains a multistreaming processor (MSP) and resides on a node module assembly. The MCM contains four processor chips (P-chips), four cache chips (E-chips), and I/O connections (two I-chips).

multistreaming processor (MSP)

For Cray X1 systems, a basic programmable computational unit. Each MSP is analogous to a traditional processor and is composed of four single-streaming processors (SSPs) and E-cache that is shared by the SSPs. See also *node*; *SSP*; *MSP mode*; *SSP mode.*

node

For Cray X1 systems, the hardware that comprises four multichip modules (MCMs) with one multistreaming module (MSP) per MCM; shared local memory that can be thought of as a cache domain; high-speed node interconnections; and system I/O ports. Physically, all nodes are the same; software controls how a node is used: as an OS node, application node, or support node. See also *application node*; *MCM*, *MSP*, *OS node*; *SSP*; *support node*.

OS node

For Cray X1 systems, the node that provides kernel-level services, such as system calls, to all support nodes and application nodes. See also *application node*; *node*; *support node*.

pointer

A data item that consists of the address of a desired item.

program counter

A hardware element that contains the address of the instruction currently executing.

scalar register

The register that serves as source and destination for operands that use scalar arithmetic and logical instructions.

single-streaming processor (SSP)

For Cray X1 systems, a basic programmable computational unit. See also *node*; *MSP*; *MSP mode*; *SSP mode*.

SSP mode (single-streaming mode)

One of two types of application modes. Programs are compiled either as MSP-mode applications (default) or SSP-mode applications. SSP-mode applications run on one or more SSPs. Each SSP runs independently of the others, executing its own stream of instructions. In contrast, compiler options enable the programmer to develop command-mode programs that run on an SSP on the support node. See also command mode; *MSP mode*.

stack frame

An element of a stack that contains local variables, arguments, contents of the registers used by an individual routine, a frame pointer that points to the previous stack frame, and the value of the program counter at the time the routine was called. A stack frame is allocated when a reentrant subroutine is entered; it is deallocated on exit.

stride

The relationship between the layout of an array's elements in memory and the order in which those elements are accessed. A stride of 1 means that memory-adjacent array elements are accessed on successive iterations of an array-processing loop.

support node

For Cray X1 systems, the node that is used to run serial commands, such as shells, editors, and other user commands (`ls`, for example). See also *application node*; *OS node*; *node*.

symbol table

A table of symbolic names (for example, variables) used in a program to store their memory locations. The symbol table is part of the executable object generated by the compiler. Debuggers use it to help analyze the program.

type

A means for categorizing data. Each intrinsic and user-defined data type has four characteristics: a name, a set of values, a set of operators, and a means to represent constant values of the type in a program.

UNICOS/mp

The operating system for Cray X1 systems.

vector

A series of values on which instructions operate; this can be an array or any subset of an array such as row, column, or diagonal. Applying arithmetic, logical, or memory operations to vectors is called vector processing.

vector length
The number of elements in a vector.

vector register
The register that serves as a source and destination for vector operations.

Index